REPLICATING EXPERIMENTS FROM EDUCATIONAL PSYCHOLOGY TO DEVELOP INSIGHTS INTO COMPUTING EDUCATION: COGNITIVE LOAD AS A SIGNIFICANT PROBLEM IN LEARNING PROGRAMMING

A Dissertation
Presented to
The Academic Faculty

by

Briana Baker Morrison

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in
Human Centered Computing

School of Interactive Computing
Georgia Institute of Technology
December 2016

REPLICATING EXPERIMENTS FROM EDUCATIONAL PSYCHOLOGY TO DEVELOP INSIGHTS INTO COMPUTING EDUCATION: COGNITIVE LOAD AS A SIGNIFICANT PROBLEM IN LEARNING PROGRAMMING

Approved by:

Dr. Mark Guzdial, Committee Chair
School of Interactive Computing
Georgia Institute of Technology

Dr. Betsy DiSalvo
School of Interactive Computing
Georgia Institute of Technology

Dr. Wendy Newstetter
School of Interactive Computing
Georgia Institute of Technology

Dr. Richard Catrambone
School of Psychology
Georgia Institute of Technology

Dr. Beth Simon
Department of Education Studies
University of California San Diego

Date Approved: November 7, 2016

To my husband, David,

my children, Shannon and Kiely,

and the rest of my extended family for putting up with me.

And for all high schools in the United States,

may you all offer at least one computing class within my lifetime.

ACKNOWLEDGEMENTS

This is a journey that I did not walk alone. I had the help of many people along the way. First and foremost, I want to thank my advisor, Mark Guzdial. He took a chance on an older student and his funding allowed me to make this journey. I owe him a great deal of gratitude for allowing me to be his apprentice for five years. I have learned so much, not only about computer science education research, but also about how to "do" research: from the framing of the question to exploring hypotheses, to completing a Guzdial chart, to always seeing the bigger picture. I would also like to thank Betsy DiSalvo and Wendy Newstetter, two of the fabulous female HCC faculty at Georgia Tech. Thank you for being passionate about education and being such excellent role models. Richard Catrambone has been an inspiration, not only because of his research, but for holding me to a higher standard. Beth Simon has been a friend and colleague for many years and I deeply appreciate her unbridled enthusiasm and prolific contributions to the CS Ed research community.

I would not have even considered this journey without two very important mentors, Dr.Johnette Hassell and Dr. Sally Fincher. Dr. H was my mentor at Tulane and she was the first to introduce me to research and plant the seed of me to pursuing a Ph.D. Sally was the catalyst almost 30 years later that ignited the spark in me to actually take the first steps towards this degree. Sally introduced me to world of CS Education research and for that I will be forever grateful. She continues to provide outstanding support and encouragement while pushing me on toward new heights.

This research would not have been possible without the phenomenal collaboration of several key individuals. Brian Dorn is a dear friend, brilliant researcher, and UNO colleague. Lauren Margulieux is my indefatigable statistics tutor and great friend. I am thrilled that through much of my journey she was able to accompany me. Adrienne Decker is my

twin spirit and together we make quite the team. I have deeply enjoyed both the laughter and tears I have shared with these three over the years and I am exceptionally thankful for their help and support.

While the Ph.D. journey is largely solitary, one cannot do it alone. I am forever indebted to the many friends who have helped me along the path. Mike Murphy and Hans Reichgelt, both from SPSU, encouraged, supported, and allowed me to pursue the journey. Many others from SPSU, too numerous to mention, encouraged and supported me during my journey. I was fortunate to begin my Tech journey with two other wonderful students, Chad Stolper and Hannah Pileggi. Though Hannah left to follow her own path, I appreciate the opportunity to live vicariously through her. I hope Chad and I will continue to be friends and colleagues for years to come. Truly, my journey would have been much more difficult without either of them. In addition, I was blessed to have awesome lab mates, especially Miranda Parker and Kayla DesPortes, to talk with, complain to, and be supported by. I also want to thank Allison Elliott-Tew, for being my Allison when I needed it and for being my friend always.

Finally, but the most significant, I want to thank my family and friends for your support, words of encouragement, and belief in my journey. There is no way I can repay all that you have done for me during this journey. Words are inadequate for the love and gratitude I feel towards you. I can only hope that I have, and will continue to, make you proud.

To Shannon and Kiely and David, thank you for being beside me throughout the entire journey. Thanks for eating way too many take-out meals when I was too busy to cook, for giving up vacations so I could study and pay tuition, for proofreading papers, for listening to me complain about being busy, and for excusing my absence for the many times I couldn't be there. This journey would not have been worth it if I did not know you would still be with me at the end of it. Your belief in me and unwavering support allowed me to persevere beyond the rough and dark spots of the road. I will always be grateful for your love and please know that you are always in my heart.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Students often find learning to program difficult. This may be because the concepts are inherently difficult due to the fact that the elements of learning to program are highly interconnected. Instructors may be able to lower the complexity of learning to program by designing instructional materials that use educational psychology principles.

The overarching goal of this research is to gain more understanding and insight into when cognitive load might be interfering with learning programming. **Cognitive load theory** (CLT), and its associated effects, describe the role of the learner's memory during the learning process. By minimizing undesirable loads within the instructional materials, the learner's memory can hold more relevant information, thereby improving the effectiveness of the learning process.

This research uses cognitive load theory to improve learning in programming. First an instrument for measuring cognitive load components within introductory programming was developed and initially validated. We have explored reducing the cognitive load by changing the modality in which students receive the learning material. This had no effect on novices' retention of knowledge or their ability to transfer knowledge. We then attempted to reduce the cognitive load by adding subgoal labels to the instructional material. This had some effect on the learning gains under some conditions. Students who learned using subgoal labels demonstrated higher learning gains than the other conditions on the programming assessment task. We also explored using a low cognitive load assessment technique (Parsons problem), to measure learning gains. This low cognitive load assessment task proved more sensitive than the open ended programming assessment tasks in capturing student learning. Students who were given subgoal labels regardless of context transfer condition out performed those in the other conditions.

In the final study I changed how students were taught a programming construct through its format and content in order to reduce cognitive load. While the changed construct was presumed to be a more natural cognitive fit for students based on previous research, the data indicated that it had no statistical difference in learning performance. Some CS educators have argued that the changed construct might harm learning performance. However, my results suggest that learning performance was not harmed, meaning that either format could be used to teach the programming construct to students without disadvantaging the learner.

CHAPTER I

INTRODUCTION AND MOTIVATION

Students often complain that learning to program is difficult. Based on failure rates [16], it's hard to disagree. But how hard is it? Is it more difficult to learn programming than calculus or physics or chemistry? We know a fair amount about math and science education, but computing and programming are relatively new fields compared to those disciplines and research on computing education is in its early stages. We should be able to learn from those disciplines on what works in teaching complex tasks and problem solving.

Educational psychology may yield fruitful insights to improving computing education. Researchers in the field of educational psychology have spent years researching how to improve both problem solving and learning of complex skills in many different fields including math and science. But few of the instructional manipulations determined to be effective in other disciplines have been empirically tested in computing. Is it possible to adapt the proven principles and effects from other disciplines and use them in computing?

The overarching goal of this dissertation is to gain more understanding and insight into when cognitive load might be interfering with learning programming. **Cognitive load theory** (CLT), and its associated effects, describe the role of the learner's memory during the learning process. By minimizing undesirable loads within the instructional materials the learner's memory can hold more relevant information, thereby improving the effectiveness of the learning process.

CLT focuses on complex cognitive tasks in which instructional control of cognitive load is critically important to meaningful learning [67]. CLT uses current knowledge about the human cognitive architecture to generate instructional techniques. This architecture consists of an effectively unlimited long-term memory which interacts with a working memory

that is limited in both capacity [13, 86] and duration [102]. New information is processed in working memory which is limited to 4 plus or minus 1 element and if not rehearsed, the information is lost within 30 seconds [36]. Long term memory contains cognitive schemas that are used to store and organize knowledge by incorporating multiple elements of information into a single element (also referred to as chunking) [114, 115]. Learning occurs if information is successfully processed in working memory and one of the following processes occur: schema creation, assimilation, elaboration, or accommodation. The information processed can form new schemas (schema creation). If new elements of information are incorporated into existing schemas this is assimilation. If the elements of lower level schema are combined into higher level schemas this increases the numbers of ever more complex schemas (schema elaboration). If the new information conflicts with existing schemas due to recurring new information which are incongruous or inconsistent with existing schemas then the existing schemas must be adapted (accommodation) [146]. Because a schema can be treated by working memory as a single element or even bypass working memory if a schema has become sufficiently automated after long and consistent practice, the limitations of working memory disappear for more knowledgeable learning when dealing with previously learned information stored in long-term memory [67].

Overcoming working memory limitations through instructional manipulations that are compatible with human cognitive architecture has been a central focus of CLT [67]. This research is interested in discerning whether cognitive load plays a factor in learning to program. First we determine a way to measure cognitive load components in the programming domain. Then experiments were constructed to test whether the results suggest cognitive overload on the participants. A low cognitive load assessment technique (Parsons problem) was used to determine if it was more sensitive to student learning gains than traditional assessment techniques (code generation). I am proposing one additional study to study another alternative to lowering cognitive load while learning to program, changing the content of a specific programming construct.

All of the experiments within this research use worked examples. **Worked examples** are specific instructional materials to be studied by the learner and are used in place of problem solving exercises. Research has shown that studying worked examples in lieu of problem solving within the domains of algebra, geometry, and chemistry improve the efficiency of learning. Worked examples lead to the same learning in less time. The final two studies use subgoal labels within the worked examples to further reduce cognitive load. Using **subgoals** within a worked example allows the user to learn a general problem solving technique and gives structure to the solution encouraging the student to "chunk" the solution and encourage schema creation. The study illustrating that cognitive overload is present when learning to program tested explanations using the modality principle. The **dual-modality** principle is the ability to effectively expand working memory by utilizing multiple modality processors.

This research develops and empirically tests techniques to lower the cognitive load while learning and assessing introductory programming concepts. The results of these experiments will help to specify when and how these educational psychology principles can be used to reduce the cognitive load when learning or assessing programming skills. This will allow us to improve student learning and allow more sensitivity in assessment techniques to determine learning gains.

1.1 Motivation

We need to pay more attention to the conditions under which programming skills are actually practiced. When a novice is presented with a new problem to solve, the learner is given a state and a set of criteria for an acceptable goal state. The learner must then apply mental operations to generate a solution, that is, a sequence of operators that enables the transition from the given state to an acceptable goal state [147]. In terms of working memory capacity, there is overwhelming evidence that generating such a solution as a novice is exceptionally expensive. It causes a high cognitive load that is extraneous to the learning

process due to the use of weak problem solving methods (e.g., means-end analysis) [147]. This bears very little relation to schema construction processes that are concerned with learning to recognize problems states and their associated actions [127]. For novice learners, learning and performing conventional tasks are different and incompatible processes [147].

Using worked examples give the learners not only a given problem state and a desired goal state, but also the example solution. Studying worked examples as a substitute for conventional problem solving allows the learner to focus attention on problem states and associated solution steps and enables learners to generalize solutions and build schemata [147]. A disadvantage of worked examples is that they do not force the learner to study them carefully. If learners only consult them whey they have difficulties in performing a problem solving task, then both the worked example and the problem solving task are being simultaneously processed in working memory resulting in a higher extraneous cognitive load [147].

Learning computing programming means both learning procedures to accomplish various goals and learning the information that is relevant to these procedures [149]. Expert programmers can easily solve problems because they are able to respond in a highly reflexive manner to abstract features of problems. When confronted with a new problem for which they have no automatic responses, they can rely on their programming knowledge to deduce a general solution. Besides developing automatic procedures, the acquisition of highly structured knowledge, or schemas, plays a significant role in learning a complex skill like computer programming.

In this dissertation worked examples are used to lower the cognitive load associated with learning and assessing learner programming skills. Specifically testing the modality of the explanations associated with worked examples, using subgoal labels to structure solutions, testing alternate assessment formats, and altering the content and structure of a specific programming task (while loops) will be examined.

1.2 Thesis Statement

Learning how to program is an activity with a high cognitive load, which I theorize is due to its high intrinsic load. Learning programming and assessing the learned knowledge involves two separate tasks: the learning activity and the assessment activity, both of which typically involve high cognitive load programming tasks. Lowering the cognitive load in either the learning or assessment activity will result in more measurable learning gains.

1.3 Research Questions

To address this thesis statement, I pose three broad research questions which will be investigated in four studies. Table 1 depicts the three questions, specific hypotheses I pose for each, and sources of data to be gathered. I discuss these studies in the remaining sections of this chapter.

1.3.1 Cognitive Load Measurement Survey

An existing survey instrument was developed for measuring cognitive load components [71]. This instrument had been tested in the domains of statistics and learning a foreign language. This study adapted the survey to introductory computer programming. Initial validation was completed and the three internal factor measurements were found to hold in the new domain. There was greater support for the intrinsic and extraneous components than the germane component which replicated previous results. The specific method used for data collection and analysis along with explanations of conclusions can be found in Chapter 3. This measurement tool is then used in the remaining studies for this dissertation.

1.3.2 Testing Modality for Code Segment Explanations

One proven strategy to reduce cognitive load is to use multiple sensory modalities for presenting the instructional material. In this study novice programming students were assigned to one of three treatments: 1) explanations given via text, 2) explanations explained

Table 1: Research Questions

Research Question	Hypothesis	Data Source
RQ1: Does altering the modality (text, oral, both) of code explanations improve student learning as measured by retention and transfer questions?	**H1A**: Students receiving oral explanations will demonstrate better retention.	• post treatment MC questions • cognitive load survey
	H1B: Students receiving both oral and text explanations will demonstrate the worst retention.	• post treatment MC questions • cognitive load survey
RQ2: Does introducing subgoals (either given or learner generated) result in better learning performance?	**H2A**: Learning activities with subgoals result in better learning performance than those without subgoals.	• problem solving assessment • cognitive load survey
	H2B: Students who generate subgoals exhibit better learning performance than those who are given subgoals.	• problem solving assessment • cognitive load survey
	H2C: A lower cognitive load assessment activity can provide evidence of learning that a high cognitive load assessment does not	• Parsons problem assessment • cognitive load survey
RQ3: What is the effect on learning performance when teaching loops that exit in the middle versus traditional loops (single entry / single exit)?	**H3A**: Exit in the middle loops will result in increased learning performance, on both low and high cognitive load assessments.	• problem solving assessment • Parsons problem assessment • cognitive load survey
	H3B: Assessment questions that use exit in the middle loop format will result in higher assessment scores regardless of instructional material format	• Parsons problem assessment • post test questions

verbally, or 3) explanations given in both and through audio simultaneously. The instructional materials were three videos explaining three introductory programming concepts (assignment and math operations, nested selection statements, and definite loops). After each video participants were asked recall and transfer questions to determine retention and transfer knowledge.

Based on previous educational psychology results, the audio only group was expected to outperform the other two groups in the recall questions. In addition, they should have reported less extraneous cognitive load than the other groups. The treatment group that received both audio and text explanations should have reported the highest cognitive load.

The actual results showed no statistically significant difference between the groups for learning, recall, transfer, or cognitive load. Details of this study along with possible explanations for the puzzling results can be found in Chapter 4.

1.3.3 Subgoal Labels

In this study participants were given a set of worked examples that were manipulated to include subgoal labels. The worked examples were designed to teach the participants how to write indefinite loops (while loops). Participants, who were introductory programming students, were assigned to treatment groups that had no subgoal labels within the worked examples (**None**), or where subgoal labels were given (**Given**), or where they were asked to generate subgoal labels (**Generate**) after initial training. In addition, each of those treatment groups were divided into two separate groups, **Isomorphic** or **Context Transfer**. One worked example − practice problem pair (WEPP) set (i.e., all three worked example − practice problem pairs) used the same context or cover story (e.g., calculate average tip). If the WEPP used the same context, the group was labeled Isomorphic. The other set contained a context changed between the WEPP. For example, the worked example may be to calculate average tip and the practice problem was to calculate the average grade. The structure of the solution was identical in both problems, only the problem story changed. This second group was labeled the Context Transfer group.

After viewing three sets of WEPP pairs the participants were given four assessment problem solving tasks asking them to write the code to solve the problem. Two of these assessment tasks were similar in structure to problems they saw in the WEPP (considered near transfer) and two of the problems were novel (considered far transfer). Participants were then given the correct solutions to those four problems and asked to group solution statements and give labels to those groups. In essence, this was asking all participants to generate subgoal labels without the identification of the grouping of statements. Finally

participants were given another new novel problem and its solution, only the solution statements were re-arranged to be in the wrong order. Participants were asked to put the solution statements into the correct order. A pre and post test on programming knowledge was also given.

It was expected that those students that generated subgoal labels would perform the best on the problem solving assessment, based on previous research results [24, 25]. It was also expected that those that were given subgoal labels would perform better on the problem solving assessment than those who received no subgoal labels. It is also possible that the contextual transfer between the worked example and practice problem may have an effect on learning gains. The contextual transfer may create too much cognitive load on those learners who were also asked to generate subgoal labels. For those learners that were given subgoal labels, the contextual transfer should allow them to generalize the subgoals across multiple examples resulting in better performance than those who saw only isomorphic problems.

Using "mixed up code" to measure learning gains is an example of using a low cognitive load assessment. This type of assessment, where the participant need not generate code from scratch but rather only select the appropriate statement in the correct order should yield a greater sensitivity to learning gains over traditional problem solving activities. Complete details of this study can be found in 5

1.3.4 Loop Strategy

The final study for this dissertation was designed to lower the cognitive load associated with learning programming by using the information learned in the previous studies and adding a new, specific, computing-only manipulation to the instructional material. Because writing indefinite loops is a difficult task for students to learn, we adopted a specific style of the indefinite loop called "loop-and-a-half" or "exit in the middle". Worked examples were created which taught the exit in the middle style of loops using subgoal labels. In the

pre / post test, isomorphic questions were developed using both the traditional and the exit in the middle style of loop questions. In this study participants were randomly assigned to one of four treatments: 1) traditional loop worked examples and traditional loop pre / post test questions, 2) traditional loop worked examples and mixed pre / post test questions (half were traditional and half were the exit in the middle format), 3) exit in the middle worked examples and traditional loop pre / post test questions, and 4) exit in the middle worked examples and mixed pre / post test questions (half were traditional and half were the exit in the middle format).

After viewing the worked example—practice problem pairs students were be asked to solve assessment problems by generating code. They were also given two low cognitive load assessments, one a traditional loop problem (identical to the one from the subgoal study) and another containing an exit in the middle solution.

It was hypothesized that those participants who learn and are assessed using the arguably more cognitively natural exit in the middle style would outperform those who learn or are assessed using the traditional loop format. Additionally the low cognitive load assessment problem should result in more sensitive identification of learning gains beyond the traditional high cognitive load code generation task. Chapter 6 contains details of this study. In general, these hypotheses were not supported by the data, but I saw no disadvantage to the use of the exit in the middle strategy.

1.4 Contributions

This dissertation will result in several unique contributions to the computer science education research community:

1.4.1 Cognitive Load Measurement Survey

The adaptation and initial validation of an instrument will allow others to measure the cognitive load components of specific instructional interventions to explain a possible reason for learning gains or reductions. By understanding the measurement of the specific

cognitive load components allows better design of instructional materials to maximize the potential for learning.

1.4.2 Modality Effect in CS

Determining if using auditory explanations for code segments improves learning in introductory programming can have an enormous impact on how future instructional material is delivered to learners. Developing design guidelines for when, if, and how to use auditory code explanations can directly impact future teaching of programming. Knowing what not to do is often the first step in designing such design guidelines.

1.4.3 Subgoal Use in CS Worked Examples

Knowing that the use of subgoals within worked examples improves learning is an easy and relatively cheap instructional intervention for instructors. Simply by adding subgoal labels to worked examples which students can study has the effect of improving learning, especially if the context changes between the worked example and practice problem.

1.4.4 Low Cognitive Load Assessment

The introduction of a low cognitive load assessment which is easily generated and graded that is more sensitive to measuring learning gains than traditional problem solving code generation assessments allows a different way to ascertain student learning. Asking students to order code statements rather than generate code statements from scratch can assess whether or not students understand the problem solution without introducing additional cognitive load.

1.4.5 Loop Style

Using an exit in the middle style loop for instruction and assessment does not necessarily improve student performance when designing loop problem solutions. However, it did not harm performance either. The data supports that either format can be used in an introductory programming course.

1.4.6 Bounded Spectrum of Learning Complexity

By evaluating the different instructional material manipulations and evaluating the effect on learning performance for each we can begin to develop a bounded spectrum for the types of learning that are difficult for learners within introductory programming. Knowing that subgoal generation produces the best student performance is useful, but only if the examples don't have contextual transfer, which then results in lower learning performance. Thus we know that generating subgoal labels with contextual transfer is more complex or "hard" than generating labels without contextual transfer. A range of instructional material manipulations can be placed on a scale indicating the internal complexity for the student based on cognitive load theory.

1.5 Dissertation Overview

The remainder of this dissertation outlines the three studies that have been completed along with their results to answer the research questions. Chapter 2 provides a review of relevant literature. Chapter 3 details the adaptation of a previously developed survey instrument to the computing discipline to measure cognitive load components. Chapter 4 discusses the modality study. Chapter 5 discusses using subgoals and Parsons problem to lower cognitive load while learning and assessing programming knowledge. Chapter 6 presents the final study involving the loop-and-a-half worked examples. The final chapter summarizes the work this dissertation represents and its contributions to the computer science education research community along with future lines of research.

CHAPTER II

BACKGROUND AND RELATED WORK

This dissertation has its grounding in several existing educational psychological principles. We begin by exploring the human cognitive architecture. To understand how we can bound the complexity of learning to program we look at **cognitive load theory** which is the underlying basis for the argument. We then look at how we can **measure cognitive load** and its components. The principle instructional method used within the experimental studies is the **worked example** format, so we will examine its place in within our framework. Within the worked examples, **subgoal labels** are utilized to reduce cognitive load so we examine the history of subgoal labels as an educational psychological principle. Finally computing specific applications of these principles are explored.

2.1 Cognitive Architecture

When learning any complex cognitive skill, two complementary processes may be distinguished. Automation offers task-specific procedures that may directly control behavior and schema acquisition involves the creation and modification of cognitive structures that provide analogies in new problem situations. Learning to program is obviously a complex cognitive skill. Unfortunately traditional introductory programming instruction often results in suboptimal automation and schema acquisition.

2.1.1 Automation

Automation leads to highly task specific procedures that may directly control programming behavior [149]. An expert, because of their knowledge of task-specific procedures, can almost automatically reformulate and decompose familiar problems into subproblems that have known solutions, and they can then effortlessly generate the programming code to

implement low-level goals.

The development of task-specific procedures is a lengthy process and it may be seen as a transition from controlled to automatic processing [149]. In the early stage of learning a complex task, the learner usually receives information about the task that may be used by general procedures or "weak" problem solving methods. In the instruction, there is usually no reference to any particular knowledge domain. This leads to poor task performance because controlled processing has the disadvantage that it works slowly and may lead to errors due to processing overload.

Consider the task of learning to read. In the beginning the learner must learn to identify individual letters and their corresponding sounds. Until this is automated, the reader must continue to identify letters to sound out words. Eventually common words become automated and the learner no longer thinks about those words but recognizes them immediately. With continued practice reading, more and more words are automated, to the point that the brain will recognize most words if only the first and last letters are correct [135]. With more practice groups of words can become automated.

Anderson identified *knowledge compilation* as the important process to make the transition from controlled to automatic processing [2, 3]. Knowledge compilation includes the incorporation of newly acquired knowledge in new task-specific procedures and the "chunking" of procedures that consistently follow each other in solving particular problems. It produces a considerable speedup in performance as more items move into a single "chunk". It also implies a reduction of the processing load as newly acquired knowledge need no longer be retrieved from memory and held active –it is automated.

As the learner progresses in knowledge through practice, the task-specific procedures are strengthened with every successful application and become automated. Automatic processing works fast, with minimal errors, and with low demands on processing capacity so that cognitive resources become available for other aspects of the task [149]. Automation is the result of practice, complex skills can only be acquired by doing them.

An effective way to present the instructional material and to shorten the training for the automation of the skill is the use of worked examples [149]. Worked examples are a type of concrete example or schema to map to new solutions. The key to using worked examples is interpreting the example by general procedures and mapping it onto the current knowledge of programming to create new solutions [149]. The use of worked examples bridges the gap between the current knowledge of the learner and facilitates the development of task-specific procedures, and eventually, automation.

Far transfer assumes an excessive decontextualization of acquired skills [149]. Automatic procedures predict transfer in so far as the procedures that are learned in the training task are identical to the procedures that are needed for performing the transfer task. In increasingly further transfer within the programming domain there is a decreasing overlap of task-specific procedures between the original task and the transfer task. So automation is no longer helpful with the transfer task. Automation cannot explain the ability to solve new problems when no task-specific procedures have been automated. Because automatic processing of certain aspects of the task makes very low demands on processing capacity, cognitive resources become available for other controlled processes.

2.1.2 Schema Acquisition

Schemas can be viewed as cognitive structures that allow particular objects, events, or activities to be assigned to general categories [149]. Schemas provide general knowledge that can be applied to particular cases. This is what allows experts to solve unfamiliar problems –they solve familiar programming tasks by using highly task-specific procedures that are automated, but they can interpret unfamiliar situations in terms of their generalized knowledge. The acquisition of several kinds of schemas is relevant to learning introductory programming [107]. A general design schema can be developed to provide abstract knowledge on the processes needed to generate a good design and overall program structure. The design schema can then be used recursively to generate a decomposition of the

problem into more and more detailed modules. This design process can continue until an automated code solution is identified for each subproblem. Ehrlich and Soloway identified programming specific schemas as programming plans [41]. Programming plans are like program templates which are a sequence of lines of code that form a hierarchy of generalized knowledge. There are high-level templates (input, process, output), medium level templates (loops with sentinel values), and low level templates (printing a value, assignment of value).

Before learning any programming, the learner has no knowledge of task-specific procedures or useful cognitive schemas available. The learner has to apply very general, weak problem solving methods to complete the programming task. When learners encounter problems while working on a learning task, the last thing they are inclined to do is further increase their already high cognitive load by processing and mentally integrating additional information from a support system [147]. As the learner continues to learn and practice the skill of programming, task-specific procedures accumulate that will increase performance on subsequent problems. In addition, schemas may be acquired that offer analogies or abstract categories of problems and solutions that may guide subsequent problem solving activities [149].

Learning may either create new schemas or adjust existing schemas to make them in line with experience. [149]. A more generalized schema may be produced if a set of solutions is available for a class of related problems. That is, a schema may be created that abstracts away from the details. A more specific schema may be produced if a set of failed solutions is available for a class of related problems. In this case the particular conditions may be added to the schema which restrict its range of use.

The process of automation slowly develops and is mainly a function of the amount of practice. The acquisition of schemas may occur rapidly but requires the investment of effort from the learner. A novice programmer requires time and practice to automate programming constructs or solutions, but schemas in form of solution templates may be

acquired quickly. This is the theory behind using worked examples with subgoals. The subgoals allow the learner to view the plan schema (i.e., loops have an initialization, test, and update), but requires time and practice before the implementation of that schema is automated.

Once useful schemas have been developed, they may then be used as analogies to generate behavior in new, unfamiliar problem situations [149]. Novices compare the current problem situation to information available in the worked examples; with increasing expertise, the current problem situation can be compared with cognitive schemas retrieved from memory. This has the advantage of flexibility, but it has the disadvantage that it works slowly and may lead to errors due to processing overload.

If the analogy repeatedly leads to the desired solutions, the schemas themselves may eventually be converted into task-specific procedures that apply to specific classes of related problems. In other words, once the learner discovers a repeated successful use for a schema for a specific type of problem, it may become automated (e.g., initialization, reading from a file, and closing the file for all file processing problems).

In order for schemas to be acquired, the learner must be presented with a wide range of different problems and solutions for the opportunity to build, generalize, or specialize schemas. The learner must pay attention and put forth effort to observe and develop the generalizations, it does not happen instinctively or automatically. Mindful abstraction is an effortful process that requires the conscious attention of the learner [149]. Therefore the instruction should encourage and even provoke the mindful decontextualization and generalization.

Acquired schemas may explain transfer by the presence of relevant knowledge from other problem solving situations and in particular, on how that knowledge is organized in schemas [149]. Both the acquisition of schemas and their use in transfer tasks requires effort and conscious attention from the learner. The availability of relevant schemas that may offer useful analogies becomes increasingly important in reaching further transfer[149].

Spontaneously noticing the analogy is a prerequisite for successful transfer in realistic problem solutions, which does not often occur in learners. Therefore, in learning schemas, it may be helpful to explicitly state that a schema is also applicable in certain transfer situations.

2.1.3 Summary

Automation and schema acquisition both play an important role in learning programming. The automation of procedures requires practice and is facilitated by the availability of worked examples, and provides identical elements that may help to solve familiar aspects of new programming problems. The acquisition of schemas requires mindful abstraction and assumes the availability of a range of problems and their solutions (i.e., worked examples) and provides analogies that may guide solutions for unfamiliar problems.

Van Merrienboër and Paas argue that automation of the more familiar aspects of programming tasks is of great importance in learning to program because it then frees up cognitive processing resources that may be devoted to both the acquisition of new schemas and the interpretation of existing schemas [149]. They continue that the processes for solving a new programming problem is then as follows: 1) familiar aspects of the task can be performed by task-specific automated procedures. These procedures can be applied fast and with minimal errors and with little or no demand on cognitive processing capacity. 2) new aspects of the task can be solve by the use of analogy. Here learned programming templates should be available to help find a solution and these schemas can be interpreted thanks to the cognitive processing resources that are freed up by automation of the more familiar aspects of the programming task.

2.2 Cognitive Load Theory

Cognitive Load can be defined as "the load imposed on an individual's working memory by a particular (learning) task" [143, p. 599]. The resultant performance for a student to learn a specific concept is directly related to how much cognitive load is used to comprehend the

material. If instruction overloads the student's working memory, then knowledge retention and any possible knowledge transfer will suffer. As designers of instructional material, it is our responsibility to ensure that we do not overload the learner's working memory where possible when presenting new material. That is, we should help ensure that students' attentional abilities are directed to key aspects of the content and learning activities that have maximum value, rather on than extraneous aspects of the material.

The central problem identified by Cognitive Load Theory (CLT) is that learning is impaired when the total amount of processing requirements exceeds the limited capacity of working memory [104]. According to Cognitive Load Theory [130, 133, 150], instruction can impose three different types of cognitive load on a student's working memory: intrinsic load (IL), extraneous load (EL), and germane load (GL). Intrinsic load (IL) is defined as a combination of the innate difficulty of the material being learned as well as the learner's characteristics [71]. A topic is considered to have a high intrinsic load if the material being learned is interconnected; that is, learning requires processing several elements simultaneously to understand their relations and interactions [132]. If interacting elements of information must be processed simultaneously in order to comprehend the task or solution, they can generate high levels of intrinsic cognitive load. When learners must concentrate on those interacting elements and attempt to mentally establish connections between them in working memory, they are actually experiencing intrinsic cognitive load. Because the intrinsic load is essential for comprehending the material and constructing knowledge structures, the instructional material must provide all the components necessary to accommodate this load without exceeding the limits of working memory capacity. Intrinsic load can also vary with the domain expertise and previous knowledge of the learner [131] in that learners with a higher level of previous knowledge may chunk the material differently than novices [18], allowing them to hold more information in working memory. Thus, the intrinsic load can change based on the learner. An element or a chunk of information for particular learners and specific tasks is determined by the organized knowledge structures

or schemas that they learners hold in their long term memory. With the development of expertise the size of a person's chunks increases and many interacting elements for a novice become encapsulated into a single element for an expert. The magnitude of the intrinsic cognitive load experienced by a learner is determined by both the degree of interactivity of the essential elements relative to the level of learner expertise within the domain [63].

Extraneous load (EL) is the load placed on working memory that does not contribute directly toward the learning of the material—for example, the attentional resources consumed while understanding poorly written text or diagrams without sufficient clarity [71]. The extraneous load is associated with cognitive processes that are not necessary for the learning and are invoked by less than optimal instructional designs. Extraneous cognitive load becomes a problem only when intrinsic cognitive load is high. If there is minimal interactivity between the elements, there is ample working memory to handle any extraneous load and learning can still occur. The effects of extraneous cognitive load (decreasing learning) can only be demonstrated when the intrinsic cognitive load is high [128].

The intrinsic load and extraneous load are the factors that can be controlled through instructional design and can thus be manipulated through experiments. The final original category is that of germane load which are the instructional features that are *necessary* for learning the material [71]. The notion of germane load was added to CLT due to unexplained empirical results to explain puzzling empirical findings that demanded a new concept. In general, cognitive load did not always interfere with learning but was always necessary for learning. No meaningful complex learning could occur without effortful cognitive processing and its associated working memory load. Because both extraneous and intrinsic loads were viewed as something to minimize and avoid, a separate type of load was introduced to account for the intentional cognitive effort leading to learning and the corresponding demands on working memory. Germane load (GL) since then has been associated with the construction and automation of organized schemas and the cognitive activities that directly contribute to learning [127].

However more recently, leading researchers are questioning whether the germane load is on the same level as that of the intrinsic and extraneous loads [63, 130, 68]. Moreno and Mayer first argued that germane load and intrinsic load occur in the same way [90]. They state that both ICL and GCL occur when less experienced learners start selecting, organizing, and integrating words and images with existing knowledge structures. This results in "essential" or "generative" processing to learn. Kalyuga argues that germane load was only added to the theory to handle unexplained empirical results and that it is impossible to experimentally manipulate the germane load [63]. He argues that germane load is actually the same thing as intrinsic load –that is, when the learner must establish connections between the elements then learning is occurring. That is what was originally referred to as the germane load. Sweller acknowledges that the three loads may not be additive but that there is some amount of mental processing necessary to learn anything, however each component may not exist at the same level. He states that germane load may be used to emphasize the amount of working memory necessary to devote to handling the intrinsic load. Kuldas et al. [68] argue for the addition of a motivational component to complement the intrinsic and extraneous as learners that are more motivated will endure more cognitive load to learn.

The key to utilizing CLT within the instructional materials is to minimize or, if possible, eliminate the extraneous load. The intrinsic load should be properly managed or designed for. Learning tasks should be selected which are not too complex relative to learner levels of expertise, but also not so simple as to no longer be sufficiently challenging and motivating within the available cognitive capacity [111]. This relates directly to selecting problems within the learners' zone of proximal development [151].

The implications of CLT lead to several different known "effects" that the learner can experience during the learning process. Principles are the underlying theories that explain how learning occurs given human characteristics. Cognitive Load Theory is a principle

explaining how learning occurs within the framework of our cognitive architecture. Empirical studies provide evidence that the theory holds. The empirical studies have found evidence that by altering the instructional design materials learning can be increased or decreased. These are known as effects resulting from specific instructional design decisions. The effects are explained within the cognitive load theory framework. Effects that have been found that are relevant to this dissertation include the following:

- worked example (problem completion) effect
- split-attention effect
- modality effect
- redundancy effect
- expertise reversal effect
- guidance fading effect
- self-explanation effect
- element interactivity effect
- transient effect

Each of these will be discussed separately.

2.2.1 The Worked Example Effect

A *worked example* provides a step-by-step solution to a problem. Learners are presented with a worked example to study, usually in place of being asked to solve the problem. The worked example effect occurs when learners who are presented worked examples to study perform better on subsequent test problems than learners asked to solve the equivalent problem [131]. Although there is no precise definition of a worked example [6], there are a number of common features. Most worked examples include a problem statement and

procedure for solving the problem. As Atkinson et al. noted, "In a sense, they provide an expert's problem-solving model for the learner to study and emulate." [6] See Figure 1 for a sample of a worked example within the introductory programming domain.

Worked examples are a means to providing the problem solving schemas that need to be stored in long-term memory [131]. They impose a relatively low working memory load as the learner does not have to employ a means-end search to find the problem solution. This means they do not have to search all their knowledge for what the next step in the solution might be. They need only look at, comprehend, and learn the next problem solving step by studying the worked example. Cognitive load theory led to studies which explicitly compared a worked example approach to learning with a problem solving approach. These studies then identified the worked example effect. The worked example effect occurs when learners perform as well or better on assessment problems when they have studied worked examples rather than solving problems. In addition, the studying of the worked examples normally requires less time than outright problem solving. So worked examples lead to equivalent or better learning in less time than traditional problem solving exercises [131].

2.2.2 The Split-Attention Effect

Some worked examples have been found to be ineffective because their format imposes a heavy extraneous cognitive load. When learners have to split their attention between at least two sources of information, both of which are necessary for learning the material, then the split-attention effect occurs. There are two necessary pieces for the split attention effect to occur: 1) the information for learning is split among into different pieces which are separated either spatially or temporally, and 2) the information in both pieces is required to learn the material and each piece would make no sense alone. Because the learner must integrate all the disparate sources of information the cognitive load to do so is unnecessarily high when the sources are separated by space or time. Having to switch focus and attention between the two sources requires information to be maintained in working memory while

searching and processing interacting elements in the linked source. This unnecessarily increases element interactivity, raising the overall cognitive load of learning. Sweller states "cognitive load theory does not distinguish between text and diagrams, text and text, or diagrams and diagrams as contributes to a split-attention effect." [129, p.98]

It should be noted that the split-attention effect is only found when the intrinsic cognitive load of the task is high. If the intrinsic load is not high, then ample working memory resources are available to compensate for the split attention. Ginns did a meta-analysis of the split-attention effect [48]. Fifty studies were included involving both spatial and temporal split-attention. The overall effect size had a Cohen's d value of 0.85 which is a large effect. No significant difference between spatial and temporal split-attention was found. The effect was weak (d = 0.28) when element interactivity was low. Consistently high effect sizes were found across math, science, and technical learning domains. To eliminate this effect when the intrinsic cognitive load of the learning task is high the sources of information should be physically integrated or synchronized.

2.2.3 The Modality Effect

The modality effect is closely related to the split-attention effect, but is only concerned when the information is being delivered through multiple sensory channels – namely the auditory and visual channels. The modality effect is one way to deal with the split-attention effect. Textual information presented in spoken form will not generate a modality effect if it merely re-describes a diagram. The information presented in the diagram and the textual information must be unintelligible by themselves. If a diagram and text are being used, both must contain information that requires learners to refer to the other source in order to enable comprehension [131].

According to the available models of multimedia learning [84, 110], cognitive processing of related text and pictures involves the selection and organization of the relevant elements of visual and auditory information resulting in a coherent unified representation.

All this is processed in the learner's working memory. CLT argues that limited working memory can be effectively expanded by using more than one presentation modality. Working memory consists three subsystems: a phonological loop, a visuospatial sketchpad and a central executive [13]. The phonological loop processes auditory information while the visuospatial sketchpad processes pictorial or written visual information. Because these are separate processes, we can assume that each have capacity and duration limitations. In some situations, like those using the modality effect, we can effectively increase the capacity of working memory by utilizing both processors.

There are some limitations to using the modality effect. Ginns conducted a meta-analysis of modality effects based on 43 different experiments [47]. The meta-analysis generally supported the positive effects of dual-modality presentations however two major moderators were found: the level of element interactivity and the pacing of the presentation. Generally only problems with a high level of element interactivity will benefit from a dual-modality presentation; however if the interactivity is excessively high the benefit of a dual-modality presentation can be lost. Strong effects of a dual-modality presentation was found only under system-paced conditions, or fixed timings.

Sweller et al. [131] list the following conditions required to obtain the modality effect:

- Diagrammatic and textual information must refer to each other and be unintelligible unless they are processed together.
- Element interactivity must be high, but not excessive.
- Auditory text should be limited. Any lengthy, complex text should be written, not spoken.
- If the diagrams are complex, cuing or signaling may be required so that learners can focus on the appropriate portion of the diagram and not be forced to search for the relevant piece.

Kalyuga [64] provides an overview of all modality studies along with instructional implications.

2.2.4 The Redundancy Effect

The redundancy effect may occur when multiple sources of information are presented and each could be understood separately without the need for mental integration [131]. The instructional materials contain, literally, redundant information. Requiring the learner to process both pieces of information and then select which is relevant increases the extraneous cognitive load. The most common form of redundancy occurs when the same information is presented in different modalities. In this case information is being processed by separate processors in working memory but still must be integrated. This additional integration work is unnecessary for the learner as presentation in a single modality would contain all the needed information for learning.

Chandler and Sweller first demonstrated the redundancy effect within a cognitive load framework [27]. Since then the effect has been replicated in many different studies covering many different disciplines [132, 29, 65]. Once again sufficiently high levels of element interactivity is required if the redundancy effect is to be observed [131].

2.2.5 The Expertise Reversal Effect

Originally the expertise reversal effect was predicted by cognitive load theory as a form of the redundancy effect which occurs when information beneficial to novice learners becomes redundant to those more knowledgeable [131]. The expertise reversal effect is a result of an interaction between a basic cognitive load effect and the level of expertise of the learner. Learners who have already acquired the knowledge and formed automated schemas are forced to "unchunk" those schemas and move back a cognitive level.

A series of longitudinal studies were conducted following groups of technical apprentices as they were trained from novice to more expert states of knowledge in engineering

areas (see [62] for an overview of these studies). Levels of learner performance and cognitive load were measured at different points to observe changes in the relative effectiveness of different instructional materials. It was demonstrated that physically integrated formats with sections of text embedded into diagrams were effective for novices, but with increasing expertise the text gradually became redundant to the content of the diagram. Tuovinen and Sweller [138] compared worked examples with learner expertise. Worked examples were better than exploration for low-knowledge learners (on using a database program) but the difference disappeared for higher knowledge learners.

Some consider the expertise reversal effect to be a form of Aptitude-Treatment-Interactions. Aptitude-treatment-interactions occur when different treatments result in differential learning rates and outcomes depending on student aptitudes [37, 38]

2.2.6 The Self-Explanation Effect

The self-explanation effect was not developed within a cognitive load theory framework, however it is presented as an addition to the worked example effect to increase learning. In self-explanation, learners engage in self-explaining connections between interacting units of information which can benefit performance [131]. It was described as "... a mental dialog that learners have when studying a worked example that helps them understand the example and build a schema from it" [34, p. 226]. The seminal paper on the topic is [30] which demonstrated that learners who process an example more deeply by explaining and providing justifications for the example learn more than students who process only the surface structures.

2.2.7 The Element Interactivity Effect

Intrinsic cognitive load is determined, in part, by the level of interactivity between essential elements of information. When element interactivity is high, adding more element interactivity associated with a high extraneous cognitive load due to suboptimal instruction design may well result in a total load well in excess of working memory capacity. The fact that

cognitive load effects tend to be obtainable only if intrinsic cognitive load is high is referred to as the element interactivity effect [130, 132]. The effect refers to the interactivity associated with both intrinsic and extraneous cognitive load. The element interactivity effect has been demonstrated for a large range of extraneous cognitive load effects, but has not been tested for all [131].

One attempt to minimize the element interactivity effect is to isolate each element and teach them one by one before teaching integration. Learning isolated elements allows students to build partial schemas that can be converted to full schemas with additional instruction that covers the interactions between elements [131]. However this method does not provide an advantage for learners with higher levels of prior knowledge in the domain. In this case they would suffer from the expertise reversal effect.

2.2.8 The Transient Effect

Excessive cognitive load can be caused by transient information. The transient information effect is a loss of learning due to information disappearing before the learner has time to adequately process it or link it with new information [131]. Most novel elements of information can be held in working (short-term) memory for a few seconds with almost all information lost after about 20 seconds unless it is intentionally rehearsed [102]. If the learner cannot process new elements of information within these time limits, the information will be lost thus hindering further processing and understanding of the learning task [63].

Whether or not transient information interferes with learning is entirely dependent on the cognitive load imposed by that information. If the information is complex and lengthy then presenting the information in a transient format, either spoken or through animation, is likely to reduce learning. If the information can be simplified by appropriately segmenting it or by ensuring that the level of learner expertise allows the information to be held and processed then a transient presentation format may not present difficulty.

2.3 Summary

The overall key to improved learning for novices is reducing the undesirable parts of the cognitive load to allow maximum memory usage for learning. One of the original assumptions of CLT is that the original three components were additive [104]; thus if the extraneous load is using the capacity of working memory, little can be devoted to the germane load. Because working memory is considered to be a fixed size [86, 14], it behooves the instructional designer to minimize the extraneous load, design appropriately for the intrinsic load, and emphasize the germane load. To accomplish this, we must be able to measure the specific load components for any pedagogical intervention. Until recently, there were no effective instruments to measure the different components of cognitive load.

2.4 Measurement of Cognitive Load

Since the discovery and identification of CLT, researchers have searched for a means to measure cognitive load. To date, this has been accomplished through indirect, subjective, and direct measures.

2.4.1 Indirect Measures of CLT

Researchers began exploring cognitive load when they saw how problem-solving activities actually interfered with learning. Computational models provided independent evidence that working memory was strained to accomplish both problem-solving and learning, so decreasing cognitive load became an important instructional design goal. Problem solving that required more searching of knowledge led to inferior learning outcomes [131]. This led Sweller and colleagues [127, 11] to develop production system models to simulate the problem solving using both high-intensive and low-intensive search strategies. Results demonstrated that high-intensive search methods required a more complex model to simulate the problem solving process.

Another indirect measure of CLT is based on learner performance indicators during the

acquisition of the material. Without a direct measure, Chandler and Sweller [27, 28] used instructional time as a proxy for cognitive load. Error rates were higher during acquisition of knowledge under conditions in which the expected time to solve was higher. This corresponded to their measurement of a higher cognitive load. Error rates have also been used to identify differences in cognitive load within problems [11, 12].

2.4.2 Subjective Measures of CLT

In 1992, Paas theorized that learners are able to assess the amount of mental effort required during learning and testing and that this 'intensity of effort' may be considered to be an 'index' of cognitive load [95, p.429]. A 9-point Likert scale ranging from very, very low mental effort (1) to very, very high mental effort (9) was used to ask learners to rate their mental effort at various points during the learning and testing cycle. Paas found that there was a correlation between self-rated mental effort and test performance. A follow-up study [97] replicated the findings and also found that the subjective ratings were more sensitive and less intrusive than an objective physiological measure also captured during this study. The 9-point scale was also found to be highly reliable [98]. The success of these initial instruments to measure cognitive load led others to adopt the subjective scale. However, during the adoption process, the wording of survey questions were changed and the term *mental effort* was changed to *difficulty* or *easy*, thus measuring something different. While subjective measures of difficulty and mental effort may be related, they are not interchangeable; difficulty does not always match effort [142].

The subjective rating scale, regardless of the wording used, has still been shown to be the most sensitive measure available to differentiate the cognitive load imposed by different instructional methods [131]. The subjective measures have also been consistent in matching performance data predicted by CLT [87]. The subjective scale has been used extensively to measure the relative cognitive load of different instructional methods with over 25 studies having used it between 1992 and 2002 [94].

Building on the initial self-rating scale, Paas and van Merriënboer developed an efficiency measure for cognitive load [96]. This efficiency measure combined both mental effort with task performance indicators. This measurement allowed instructors to determine whether specific pedagogical interventions yielded high or low instructional efficiency results. Over 30 cognitive load theory related studies have used this efficiency measure [142]. However the adoption of this measurement tool also resulted in changes from the original use: measuring the mental effort changed the efficiency of learning. Obviously measurement in and of itself adds to cognitive load for participants. Also, *when* the mental effort was measured changed what the efficiency was measuring. For those studies where mental effort was measured immediately following the acquisition phase and prior to the testing phase, training efficiency was measured. For those studies where mental effort was measured after test performance, learning efficiency was measured.

2.4.3 Direct Measures of CLT

Two basic means of measuring cognitive load through direct measures have been used: using a dual task and physiological measurements. A secondary or dual task study requires learners to engage in an additional cognitive activity that is secondary to the primary task of learning. If a higher cognitive load is required for the primary task, performance on the secondary task will suffer. Usually the secondary task is quite dissimilar and requires less working memory than the primary task, such as recognizing when a letter changes color or when a specific tone is heard. Examples of cognitive load studies using the dual task methodology can be found in [20, 29, 141]. In general, CLT research has made far less use of the dual task method than the subjective method for measuring cognitive load. However the advantage of using a dual task method is that it can provide an almost continuous measure of cognitive load during a task.

Physiological measures can also provide a continuous measure of cognitive load during a task. Researchers have used measurements of heart rates [98], pupillary response [140],

EEGs [5], and eye tracking [144, 139]. Others have even advocated using fMRI [152]. In general the results support measuring CLT using physiological measures, but the studies have only been run in laboratory settings due to the requirement of specialized equipment. This raises questions about the ecological validity of such studies and highlights the impracticality of using these approaches in most applied educational research.

2.4.4 Critique of CLT Measurements

The basic question of a cognitive load measurement method is whether it is valid, reliable, and practical. Subjective rating scales are simple and practical to use and have proven reliable through repeated use. However, they only deliver a one point post hoc assessment of the cognitive load imposed by the learning task. It remains unclear which of the specific aspects of the learning situation caused the level of cognitive load reported by the student. Although it is assumed that learners are able to be introspective about their own cognitive processes and quantify their perceived mental load during learning, the measures are unable to provide information regarding which processes caused the perceived amount of mental load. It is also not possible to determine which of the three types of load (IL, EL, or GL) originated the report of mental effort.

Objective methods yield data for more than a single point in time by providing information about instantaneous, peak, average, accumulated, and overall load [5]. However these measurement techniques are far from practical for the majority of any large scale study based on the specialized equipment needed. Objective measurement methods also suffer from the inability to determine which type of load was responsible for the resultant physiological changes.

Several researchers have attempted to distinguish between and measure the different types of cognitive load. Ayres attempted to keep the extraneous cognitive load (EL) constant between treatments thus attributing the differences to a change in intrinsic load (IL)

[10]. DeLeeuw and Mayer used a mixed approach (both subjective measures and a secondary task method) to investigate if different instruments could measure the three loads separately [39]. The results indicated that different measures do tap into different processes and show varying sensitivities.

A widely used multidimensional scale is the *NASA Task Load Index (NASA-TLX)* [56], which consists of six subscales that measure different factors associated with completing a task. An overall measure of task load is achieved by combining the subscales of the NASA-TLX. One of those subscales is mental demand/load. However the NASA-TLX was designed for use with interface designs and specifically for the aeronautical industry. In an attempt to measure the different cognitive load categories, Gerjets, Scheiter, and Catrambone selected three items from the NASA-TLX associated with task demands [45, 46]. The researchers argued that the three items selected (mental and physical activity required, effort to understand the contents, and navigational demands of the learning environment) could be mapped to the intrinsic, germane, and extraneous loads, respectively. The test manipulated the complexity of worked examples. There was broad agreement with the test performance data in that groups with the highest learning outcomes reported the lowest cognitive load. However there was no corroborating evidence that the three measures corresponded to the different types of cognitive load as proposed.

However in 2013, Leppink et al. [71] developed an instrument specifically for measuring different types of cognitive load which consists of a ten question subjective survey. (This survey will be referred to as the *Cognitive Load Component Survey*.) The researchers developed the questions and tested them using a set of four studies. The overall purpose of the studies was to compare their instrument as a measurement tool for the three types of cognitive load to other existing subjective measurement tools. The first study used exploratory analysis to determine if the questions developed did indeed load onto the three types of cognitive load (IL, EL, and GL). The second study used confirmatory factor analysis (CFA) to test the existing measurement tools [10, 31, 109, 95] for measurement of

specific cognitive load factors. This study revealed that none of the existing survey tools adequately separated the three types of cognitive load in that each had significant cross-loading between factors. The newly developed Cognitive Load Component Survey was also tested using confirmatory factor analysis in the third study. The final study then used the Cognitive Load Component Survey to examine the effects of experimental treatment and prior knowledge on the cognitive load components and learning outcomes of students within a statistics course. The results of the final study were consistent with outcomes based on CLT.

Leppink et al. [72] recently extended their 2013 work by adapting the survey instrument to another domain, that of learning languages, and replicated their analyses. These new findings reinforce the strong support for the survey measuring both intrinsic and extraneous load, but found less support for the direct measure of germane load. This fits with the more recent trend toward eliminating germane as a separate cognitive load component.

2.5 Worked Examples

Previously the worked example effect was explained which is the learning effect that occurs when worked examples are used as the instructional materials during the acquisition of knowledge phase. Worked examples are the format for all of the instructional materials used within this dissertation and deserves further elaboration. Worked examples give learners concrete examples of the procedure being used to solve a problem. According to Atkinson et al., [6], research into worked examples has a long history going as far back as the 1950s.

The lack of guidance and modeling during problem solving imposes a high cognitive load which may result in either directing the attention away from those aspects of the task that are important in learning or in completely losing one's way [127]. Such ineffective practice and cognitive overload may eventually lead to decreased motivation and a further

impairment of performance [149]. The direct availability of useful worked examples during practice was found to be far more effective than the conventional use of illustrative examples. Using worked examples shortens the acquisition phase, reduces the number of errors made during acquisition, and improves both near and far transfer [35].

Students can use worked examples as blueprints to map onto their program solutions which supports automation. Student can also generalize from the worked examples to learn new programming principles, design techniques, and programming templates so that they also support schema acquisition. Using worked examples may also encourage mindful abstraction; at a minimum the students are continuously presented with materials from which they can abstract away from.

Sweller and Cooper [126] used algebraic manipulations to first test worked examples using the cognitive load theory. They found improved test performance by the worked example group on problems similar to the acquisition problems, but they failed to find evidence of transfer. In a follow up study [35] they used algebraic manipulations and word problems along with extra learning time to prove a worked example effect for transfer problems. Remember that when first learning, every step of the process is new and requires mental processing. Once the process is learned, or automized, less mental processing is required. Automation means working memory resources are available for other activities during problem solving [131]. However automation takes place slowly and requires substantial acquisition time. The Cooper and Sweller 1987 experiments gave the learners the extra acquisition time to allow for automation whereas the initial experiments did not. Cooper and Sweller concluded that within any complex domain, significant acquisition time is required to automate the necessary problem solving steps to demonstrate transfer. Worked examples were found to accelerate this process as compared to a problem-solving approach.

Other worked example studies have been completed in the areas of statistics [95] and geometry [97].

Worked examples can be presented in various formats: blocked, a series of worked examples followed by a series of practice problems; pairs, a worked example followed by a practice problem; and stand alone, worked examples to use as reference materials for future problem solving. Most worked example studies utilize the alternation or pair strategy: learners study a worked example and are then asked to immediately solve a similar problem. This is as opposed to the control group which would be asked to solve both problems. The methodology of pairing a study problem along with solving a problem was first used by Sweller and Cooper [126] as a means to improve learner motivation. They felt that if the learner was immediately asked to solve a problem similar to one they had just been presented with they would be encouraged to study and learn the worked example rather than merely skimming through the surface features of the example. To test the effectiveness of the alternation strategy, Trafton and Reiser [137] performed a study that included blocked practice and alternating practice. They found that for an example to be most effective it had to be accompanied by a problem to solve. The most efficient method of studying examples and solving problems was to present a worked example and then immediately follow this example by asking the learner to solve a similar problem.

Eiriksdottir and Catrambone argue that learning primarily from worked examples does not inherently promote deep processing of concepts [42]. While it may result in better initial performance because examples are more easily mapped to problems, it is less likely result in the retention and transfer [42]. When studying examples, learners tend to focus on incidental features rather than the fundamental features because incidental features are easier to grasp and novices do not have the necessary domain knowledge to recognize fundamental features of examples [30]. For example, when studying physics worked examples, learners are more likely to remember that the example has a ramp than that the example uses Newton's second law [30]. A focus on incidental features leads to ineffective organization and storage of information that, in turn, leads to ineffective recall and transfer [18]. The key to designing good worked examples is in reducing the extraneous cognitive

You recently just graduated from high school. Many of your friends and relatives wanted to congratulate you on this wonderful accomplishment and sent you checks as presents. You now have all of these checks and you want to deposit them into your bank account. But first, you need to determine how much money you are depositing so you can fill out the form properly.

If you have checks in the following amounts, how much will you be depositing?

Check 1 : $25.00	Check 4: $55.00	Check 6: $20.14
Check 2: $50.00	Check 5: $60.00	Check 7: $20.00
Check 3: $35.00		

Solution

```
checks = [25, 50, 35, 55, 60, 20.14, 20]
sum = 0
lcv = 0
WHILE  lcv < length(checks) DO
   sum = sum + checks[lcv]
   lcv = lcv + 1
ENDWHILE
PRINTLN sum
```

Figure 1: Worked Example for Writing a Loop

load. Atkinson et al. [6] give guidelines for creating effective worked examples.

2.6 Subgoal Labels

To promote deeper processing of worked examples and, thus, improve retention and transfer, worked examples have been manipulated to promote subgoal learning. Subgoal learning refers to a strategy used predominantly in STEM fields that helps students deconstruct problem solving procedures into subgoals, functional parts of the overall procedure, to better recognize the fundamental components of the problem solving process [8]. Subgoals are the building blocks of procedural problem solving and they are inherent in all procedures except the most basic. Subgoal labeling is a technique used to promote subgoal learning that has been used to help learners recognize the fundamental structure of the procedure being exemplified in worked examples [24, 23, 25]. Subgoal labels are function-based instructional explanations that describe the purpose of a subgoal to the learner. For example,

in the problem in Figure 1 for the first two lines of code the subgoal label might read "Initialize Variables." This label provides information about the purpose of that subgoal and the function behind the steps within it. Studies [7, 9, 24, 23, 25, 77, 74] have consistently found that subgoal-oriented instructions improved problem solving performance across a variety of STEM domains, such as programming (e.g., [77]) and statistics (e.g., [25]).

Studies have found that giving subgoal labels in worked examples improves performance while solving novel problems without increasing the amount of time learners spend studying instructions or working on problems (e.g., [77]). Subgoal labels are believed to be effective because they visually group the steps of worked examples into subgoals and meaningfully label those groups [8]. This format highlights the structure of examples, helping students focus on structural features and more effectively organize information [6].

By helping learners organize information and focus on structural features of worked examples, subgoal labels are believed to reduce the extraneous cognitive load that can hinder learning but is inherent in worked examples [6]. Worked examples introduce extraneous cognitive load because they are necessarily specific to a context, and students must process the incidental information about the context even though it is not relevant to the underlying procedure [130]. Subgoal labels can reduce focus on these incidental features by highlighting the fundamental features of the procedure [6]. Subgoal labels further improve learning by reducing the intrinsic load by providing a mental organization (i.e., subgoals) for storing information.

Subgoal labels that are independent from a specific context have been the most effective type of subgoal labels in the past [26, 25]. Catrambone found that learners who were given labels that were abstract (e.g., Ω) and had sufficient prior knowledge performed better than those who were given labels that were context-specific (e.g., isolate x) on problem solving tasks done after a week long delay or in problems that required using the procedure differently than demonstrated in the examples [25]. Catrambone explained this exception by arguing that learners with sufficient prior knowledge were able to correctly explain

to themselves the purpose of the subgoal and that by self-explaining the function of the subgoal–the self-explaining presumably due to the abstract label–was more effective than providing labels.

2.7 CS Specific Studies

In considering using these principles within the computing domain, it should be noted that computer programs are very complex. Asking students to generate new programs may cause a high working memory load which is intensified by the need for learners to search for, and refer back to, equally complex model programs. This creates a very high extraneous cognitive load. By presenting learners with solved worked examples to study prior to problem solving so that they do not have to search for solutions is a way to reduce that extraneous cognitive load.

The first research in using worked examples in CS is [103, 105] which involved the LISP programming language. In a series of studies designed for implementing and testing a cognitive tutor, Pirolli et al. used a form of worked examples to test student knowledge and learning capabilities. The result of the studies showed that students who studied the worked examples, especially with self-explanation skills, performed better than those who simply solved problems or had poor self-explanation skills.

2.7.1 Measuring Cognitive Load in CS

Other than work by this author, the only other known attempt to measure cognitive load within computing was done in [81, 82]. Mason and colleagues surveyed students in introductory programming courses offered by Australian universities and asked about mental effort. Participants were asked to rate their own levels of mental effort on each of the three components of cognitive load using a 9 point Likert scale. They were also asked to estimate the levels of mental effort on each component experiences by an average student in their introductory programming course and that experienced by a student in the "bottom 10% of performance" in their course. However their survey questions were never published or

validated.

2.7.2 Worked Examples in CS

Some research has been done directly on using worked examples within the computing domain. Van Merriënboer conducted the first extensive study on using a worked example completion format using introductory computer programming problems [148]. A follow-up study [145] used groups that generated programs to those that completed programs. When using a completion strategy, the presentation of new information and programming practice were linked to incomplete programs and learners were only required to complete the partial solutions. The generation strategy presented both model programs and generation assignments, with the model programs considered the worked example. However, the students were not required to study the model program before beginning generation. The completion group had better post performance.

Casperson and Bennedsen [22] present a case for using worked examples in CS but no empirical evidence validating the use with student learning. Skudder and Luxton-Reilly [116] present sample worked examples that might be used in an introductory programming class with reasoning on why it might be beneficial, but no empirical evidence on use within an actual class with students. Gray et al. [52] present a detailed set of suggestions for implementing faded worked examples (similar to program completion) for an introductory programming course in C++. They decompose the task of programming in to components whose cognitive load can be adequately handled by the students. The decomposition is based on the abstract algorithmic dimensions and the associated concrete programming constructs. The authors provide concrete fully worked examples for all of the design construct and implementation-construct pairs. However, once again, no empirical evidence exists on using the examples with students in either a laboratory or classroom based study.

Finally, Garner [44] designed and implemented a Code Restructuring Tool (CORT) to implement code completion tasks based on cognitive load theory and worked examples.

The tool was designed to teach the Visual Basic programming language. An empirical quasi-experimental study was run that showed students receiving the intervention spent less time solving problems but had no learning gains over the control group.

2.7.3 Subgoals in CS

The only known work using subgoals in computing is [75]. This work used subgoal labels to teach learners how to develop a mobile application using MIT's Android App-Inventor. Participants were given a video using subgoal labels as callouts to provide structure to the solution process. The subgoal group attempted and completed successfully more subgoal steps of the assessment tasks, in addition to completing the tasks quicker than the control group. The subgoal group also successfully completed more tasks on a retention task tested one week later. In a second study involving a think-aloud protocol while completing the app building task the subgoal group outperformed the control group. In addition, Margulieux et al. found that the student vocabulary included the subgoals.

The app building task done in both of these studies involved only sequential steps. No selection statements or repetition statements were required to solve the tasks. It may be that the addition of selection statements and / or repetition statements add enough cognitive load to make subgoal labels even more important in structuring the solution.

2.7.4 Parsons Problems

One way to make the learning of programming more efficient and effective is to reduce the amount of time that learners struggle with syntax errors. One approach is to use Parsons problems [101] in which correct code is broken into code fragments that have to be put in the correct order with the correct indention. There are several variants of Parsons problems such as including unnecessary code as distractors [40]. Work in this area [40] has found that Parsons problems scores significantly correlate with code writing scores. Parsons problems are simpler than writing code, e.g., students cannot get syntax errors. It has a lower cognitive load because students do not have to focus on issues like syntax while practicing

meaning and sequencing within problem solving. This means that Parsons problems might be a more efficient way to practice than the traditional approach, hours of writing code.

2.7.5 Metrics in Computing

The best known of the early approaches to software complexity measurement were those of McCabe and Halstead [21]. They are probably still the best known and most widely used. They have also been the subject of a large amount of empirical and theoretical research. However their dominance does not appear to be based on any impressive empirical success, but rather on the lack of any validated 'rivals' [21]. While there is no accepted definition of software complexity, Cant et al. define it as *The cognitive complexity of software refers to those characteristics of software which affect the level of resources used by a person performing a given task on it.* Notice that the definition is based upon the level of knowledge of the person doing the task and allows the complexity to be operationalized as a measurable variable. Cant et al. proposed the Cognitive Complexity Metric (CCM) in an attempt to quantify the cognitive processes involved in program development, modification, and debugging [55]. Operational definitions for each of the factors in the metric that are believed to influence each process are given. Some of the factors in the CCM are operationalized by drawing upon existing literature, but many definitions are placeholders for future empirical studies. Because the metric is determined based on the expertise and skill of the person performing the task it can be used with both experts and novices.

The CCM focuses on the processes of *chunking* (understanding a block of code) and *tracing* (locating dependencies within the code). We know that novices and experts have different size chunks, thus examining the metric for the contributing factors will let us understand how chunking might affect the overall complexity measure. Novices typically are not concerned to finding dependencies in large code projects, but do have to determine the dependencies within their own code. Chunking is defined as the process of recognizing

groups of code statements (not necessarily sequential) and recording the information extracted from them as a single mental symbol or abstraction [21]. Cant et al. admit that "it is difficult to determine exactly what constitutes a chunk since it is a product of the programmer's semantic knowledge, as developed through experience." For purposes of the CCM it is defined as a block of statements that much occur together.

Shneiderman and Mayer [113] suggest that the chunking process involves utilizing two types of knowledge: semantic and syntactic. Semantic knowledge includes generic programming concepts from loops all the way up to sorting algorithms. This semantic knowledge is stored in long term memory independently of programming language. The information is also systematically organized in a hierarchical structure so that related concepts are aggregated into a single concept at a higher level of abstraction. Syntactic knowledge is programming language specific and allows semantic structures, implemented in that programming language to be recognized. By chunking, syntactic knowledge is used to convert code into semantic knowledge [21].

To compute the complexity of C_i of chunk i, the following equation is used:

$$C_i = R_i + \sum_{j \epsilon N} C_j + \sum_{j \epsilon N} T_j$$

where R_i is the complexity of the immediate chunk i, C_j is the complexity of sub-chunk j, and T_j is the difficulty in trading dependency j of chunk i. R is defined as

$$R = R_F(R_S + R_C + R_E + R_R + R_V + R_D)$$

and T is defined as

$$T = T_F(T_L + T_A + T_S + T_C)$$

Each term on the right represents a specific factor that is thought to influence the chunking or tracing processes. See Figure 2 for an overview, but each factor is described below.

Term	Description
R_F	Recall speed (familiarity)
R_S	Size of chunk
R_C	Type of control structure chunk is embedded in
R_E	Difficulty of understanding (complex expressions)
R_R	Recognizability of chunk
R_V	Effects of visual structure
R_D	Dependency disruptions
T_F	Dependency familiarity
T_L	Localization
T_A	Ambiguity
T_S	Spatial distance
T_C	Level of cueing

Figure 2: Factors of CMM. R_x terms affect chunk complexity. T_x terms affect tracing difficulty

2.7.5.1 Chunk Complexity

$\mathbf{R}_F$ **(*Chunk familiarity*)** captures the speed at which the programmer is able to understand a given a chunk of code. This represents how automated the chunk is within the programmer's memory. An expert with many automated task-specific procedures would be able to immediately recognize a summing loop, while the novice may require multiple readings of the lines of code before that is recognized. Notice that this is the multiplier term within the complexity measure. This factor is thought to have the largest influence on the chunk complexity term.

$\mathbf{R}_S$ **(*Size of chunk*)** represents both the structural size (how many lines of code) but also the "psychological complexity of identifying a chunk where a long contiguous section of non-branching code must be divided up in order to be understood." That is, where the breaking point of one chunk ends and the next begins. Notice that this term is also effected by short-term memory constraints, or how much the programmer can hold in memory at one time. While we know that experts chunk in larger sizes than novices, experts can also more easily recognize the beginning and end of chunks without visual cues such as white space.

$\mathbf{R}_C$ **(*Control structures*)** represent the type of control structure in which the chunk is embedded. Conditional control structures such conditional statements and repetition structures require the programming to comprehend additional boolean expressions and are more difficult than sequential statements.

$\mathbf{R}_E$ **(*Boolean expressions*)**, actually the complexity of the boolean expressions in a chunk depends heavily on its form and the degree to which they are nested. Introductory programming instructors understand that long, nested boolean expressions are difficulty for novices to decipher. Whereas an expert can easily ignore pieces of the boolean expression through short circuiting or by applying De Morgan's Law.

$\mathbf{R}_R$ **(*Recognizability*)** captures the notion of whether the chunk adheres to standard programming practices with variable names and format. Empirical studies have shown that the syntactic form of a program can have a strong effect on how a programmer mentally abstracts during comprehension [55].

$\mathbf{R}_V$ **(*Visual structure*)** is determined by how visual boundaries influence chunk identification. Is the chunk bounded by whitespace or comments? Is it a control structure with well indented code? Each of these things signal to an expert a chunk boundary while novices are not likely to learn this without considerable practice.

$\mathbf{R}_D$ **(*Dependency disruptions*)** attempts to capture the idea of how many disruptions to comprehension there are based on having to resolve dependencies. If the programmer must look up variable declarations or APIs that are external to the chunk that is a dependency disruption. Again this relates to working memory in terms of how many dependencies can be held in working memory at one time. An expert programmer may not need to look up common APIs which she has internalized into her schema, but a novice may need to look up every dependency.

2.7.5.2 *Tracing Difficulty*

Most code segments contain references to variables or functions or APIs which are defined elsewhere. These references represent dependencies which must be resolved before the code segment can be understood. That is what this term in the CCM attempts to capture.

$\mathbf{T}_F$ **(*Familiarity*)** is similar to chunk familiarity. This represents whether or not the programmer is already familiar with the dependency (e.g., it is a know API) versus if the dependency is unknown. Expert programmers will have much more familiarity with pre-defined dependencies within the programming language (e.g., API libraries) than a novice.

$\mathbf{T}_L$ **(*Localization*)** represents the degree to which a dependency may be resolved locally. Cant et al. proposed three levels of localization: embedded –the dependency is resolvable within the same chunk, local –the dependency is resolvable within the module boundaries, and remote –the dependency is defined outside the module boundaries.

$\mathbf{T}_A$ **(*Ambiguity*)** is a binary factor indicating whether or not there are multiple lines of code that depend upon or are effected by the current dependency. If the dependency is located with the condition of a selection statement then understanding the dependency is crucial for determining the control path. As opposed to a dependency within a print statement that is unimportant for the tracing task; there is no other line that is effected by that dependency.

T_S **(*Spatial distance*)** is the distance between the current line of code and its dependency which affects the difficulty of tracing. If the line of code containing the dependency is far away from the dependency resolution spatially then the difficulty of tracing increases. Note that with today's development environments and the ability to click on a line of code within an integrated development environment and have the dependency immediately open in another window lessens the impact of this factor.

$\mathbf{T}_C$ **(*Level of cueing*)** is a binary term that represents whether or not a reference is considered "obscure". This term appears to be related to the effect of visual structure on a chunk's complexity (R_V. Clear boundaries of chunks are likely to play a role in the value

of this term as well.

In looking at the CCM we begin to discover the types of things that make learning programming complex. Specifically for the chunk complexity, the ideas of chunk familiarity, chunk size, and the type of control structure that contains the chunk have a direct impact on the complexity of the code. Only through continued exposure and practice can students increase their familiarity and recall (automation). Through repeated successful practice, like that provided by worked example-practice pairs, schema elaboration occurs thus increasing the size of each chunk. Subgoal labels can also help with this process by structuring the solutions into definable chunks.

2.8 Conclusion

We have examined how learning occurs through the automation and schema creation. Cognitive Load Theory is based upon the limitations of the human cognitive architecture. By specific manipulations of instructional materials we can manage the cognitive load placed on working memory during the learning process. Specific ways to measure cognitive load were reviewed. Relevant literature exploring the empirical use of worked examples and subgoal labels were discussed. Finally, work done within the computing domain were reviewed.

CHAPTER III

COGNITIVE LOAD MEASUREMENT STUDY

Because we want to reduce both the overall cognitive load and specifically the undesirable cognitive load components to improve learning, we need a way to determine if we have accomplished this goal. To do this, we must be able to measure the cognitive load associated with instructional materials; specifically we need to measure the intrinsic and extraneous loads that learners experience with the learning intervention. The first study of this dissertation adapted an existing cognitive load measurement tool to introductory computing and established initial statistical validity.

In 2 various cognitive load measurement techniques were presented. The most recent is the Cognitive Load Component Survey [71, 72]. This study adapts the Cognitive Load Component Survey for use in an introductory computer science context. Details of how the instrument was adapted to a different discipline and the results of measuring the cognitive load factors of specific lectures are presented [1].

3.1 Study Method

Development and initial validation work on the original Cognitive Load Component Survey identified a set of three underlying dimensions (factors): 3 items related to intrinsic load, 3 items measuring extraneous load, and 4 items measuring germane load [71, 72]. Participants respond to each item using an 11-point semantic differential scale from 0 to 10 anchored at "0-not at all the case" and "10-completely the case". Multiple CFA studies using data from different undergraduate statistics lectures confirmed that this three factor model consistently performs well [71, 72].

[1] This work was done with the assistance of Brian Dorn who completed all the statistical analysis and statistical results.

Given the consistent findings related to using the cognitive load questionnaire in statistics lectures, the questionnaire was adapted and its applicability for computer science was verified. Since the underlying cognitive load theory dimension tied to each factor should be independent of any particular discipline, the method for validating this adaptation hinges on verifying that the original factor model holds true for the newly reworded items in the new disciplinary context of computing. That is, the structure of items on the questionnaire should be robust to slight alterations in question wording better suited to terminology used in computer science coursework.

Similar to the changes done by [72], the wording was changed in a total of three questions. The original question 2 was "The activity covered formulas that I perceived as very complex." This was changed to "The activity covered program code that I perceived as very complex." In questions 2 and 9 the word "formulas" was changed to "program code". In question 8, the word "statistics" was changed to "computing / programming". These word changes were piloted with a small group of students to determine if they were understood by participants and were appropriate for the concepts addressed during lectures. The final modified instrument instructions and items are provided in Figure 3.

As posited by Leppink et. al. [71], items 1, 2, and 3 measure the intrinsic load (IL); items 4, 5, and 6 are the extraneous load (EL) factors; and items 7 through 10 measure the germane load (GL). Note that the wordings for items 1 through 6 are negatively worded, in the sense that a response of "10-completely the case" indicates a very high detriment to learning. This is in comparison to items 7 through 10 which are positively worded. Because items 1 through 6 are measuring factors (IL and EL) that we want to minimize, the higher the response indicates that more working memory is being allocated toward undesirable components; while higher scores on items 7 through 10 indicate a desirable use of working memory.

Instructions: *All of the following questions refer to the lecture that just finished. Please respond to each of the questions on the following scale by circling the appropriate number (0 meaning not at all the case and 10 meaning completely the case):*

1. The topics covered in the activity were very complex.
2. The activity covered program code that I perceived as very complex.
3. The activity covered concepts and definitions that I perceived as very complex.
4. The instructions and/or explanations during the activity were very unclear.
5. The instructions and/or explanations were, in terms of learning, very ineffective.
6. The instructions and/or explanations were full of unclear language.
7. The activity really enhanced my understanding of the topic(s) covered.
8. The activity really enhanced my knowledge and understanding of computing / programming.
9. The activity really enhanced my understanding of the program code covered.
10. The activity really enhanced my understanding of the concepts and definitions.

Figure 3: CS Cognitive Load Component Survey

3.1.1 CS Cognitive Load Component Survey

The CS Cognitive Load Component Survey (CS CLCS) was administered twice during the term of an introductory course in computing using Python designed for non-CS majors that utilizes a media computation context. The students were declared majors in Liberal Arts (mostly literature, public policy, and international affairs), Business, and Architecture. Data was collected from two different sections of the course. Both sections were taught by the same instructor and covered the same material on the days of collection. The first dataset (Lecture 1) was collected mid-way through the course. Students had already completed several assignments involving text and list manipulations and image processing. The first dataset was collected after a lecture on generating HTML for web pages using Python functions. The second dataset (Lecture 2) was collected in the last 20% of the course and followed the initial lecture on sound processing. Students saw visualizations of different sounds, heard an explanation for how sound is digitized, and saw demonstrations of code

for manipulating volume.

Surveys were paper-based and distributed at the end of each lecture. Approximately 10 minutes at the end of class was allocated to explaining the purpose of the survey, distribution and collection of the instrument. The survey items were presented in three different orders (see Table 2). The three versions were put into randomized order so that people sitting next to each other were not necessarily answering the questions in the same order. This mitigates ordering effects within the questions that might skew the results.

3.2 Data Analysis

After each collection of data, surveys with obvious patterns (e.g., all 5's, zig-zag responses) were filtered out to ensure participants had appropriately considered each question prompt. For Lecture 1, only one invalid survey was removed from processing, and for Lecture 2, two invalid surveys were eliminated. The survey yielded a total of 156 valid responses from both sections following lecture 1 and a total of 117 valid responses following lecture 2.

Each data set was then analyzed using confirmatory factor analysis (CFA). CFA allows the researcher to propose and test underlying relationships between items on a survey [30] [19]. It is commonly used within instrument validation to determine whether groups of survey questions that theoretically relate to one another also relate statistically. CFA uses the co-variance matrix of item responses to investigate the degree to which a specified model explains the observed variation in the data set. It produces a set of model fit statistics and parameter estimates that are interpreted to evaluate how well a proposed model captures these relationships.

Table 2: Question Ordering

		Count in Lecture	
order	Item Order	L_1	L_2
A	1, 7, 4, 2, 8, 5, 3, 9, 6, 10	53	40
B	6, 10, 9, 3, 5, 8, 2, 7, 1, 4	52	43
C	9, 3, 6, 8, 2, 4, 10, 5, 7, 1	51	34

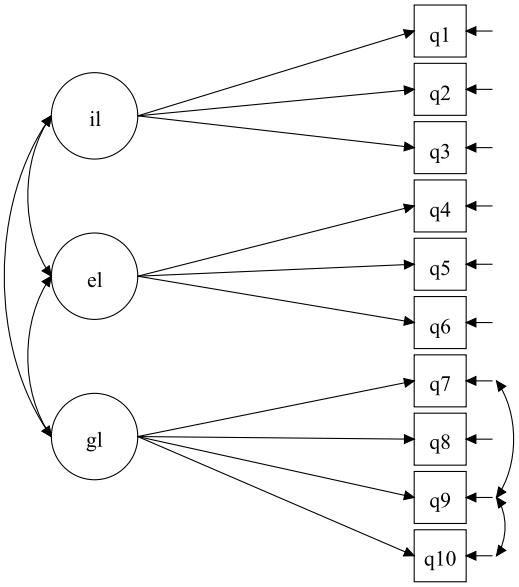

Figure 4: Factor Model

This data was analyzed using MPlus for confirmatory factor analysis. The factor model found by the earlier work (see Figure 4) [71] was replicated. In this model each item serves as an indicator of a single factor (i.e., it loads on only one factor), and the error terms of two pairs of items to co-vary (q7 & q9, q9 & q10) due to similarities in the wording of question stems. While Leppink et al. [71] failed to indicate the statistical estimation algorithm used in their CFA on the original questionnaire, it is probable that maximum likelihood (ML) was used as it is the default in most CFA software packages and is the one most commonly used [19, 58]. Thus, we employed the ML estimator in our analysis. The results of the CFA analysis for the adapted questionnaire are outlined in the following section.

Table 3: Factor Loadings and Reliability (Lecture 1)

Factor/Item	Factor Loading	Std Err	p-Value
IL - Intrinsic Load (α = 0.85)			
Q1	0.75	0.043	< 0.001
Q2	0.93	0.031	< 0.001
Q3	0.76	0.040	< 0.001
EL - Extraneous Load (α = 0.80)			
Q4	0.92	0.030	< 0.001
Q5	0.71	0.046	< 0.001
Q6	0.69	0.048	< 0.001
GL - Germane Load (α = 0.92)			
Q7	0.91	0.023	< 0.001
Q8	0.86	0.027	< 0.001
Q9	0.87	0.030	< 0.001
Q10	0.80	0.034	< 0.001
Residual Covariance			
Q7 with Q9	-0.19*	0.157	0.230
Q9 with Q10	0.40*	0.091	< 0.001

*denotes a correlation rather than a loading

Table 4: Factor Correlations (Lecture 1), all significant at $p \leq 0.001$

	IL	EL	GL
IL	1.0	0.435	-0.282
EL		1.0	-0.749
GL			1.0

3.3 Results and Contributions

3.3.1 Lecture 1

Fit statistics provided by confirmatory factor analysis on data collected following lecture 1 (N=156) indicated that the model specified in Figure 4 fit the data well ($\chi^2(30) = 36.92$, $p = 0.18$; $RMSEA = 0.04$, $pclose = 0.67$; $CFI = 0.99$; $TLI = 0.99$). The χ^2 test yielded a non-significant p-value, indicating that the observed co-variance matrix was not significantly different from the expected matrix specified by the model. Further, this model exhibited other fit statistics meeting or exceeding cutoff criteria for well-fitting models specified by Hu and Bentler [57]. The root mean square error of approximation (RMSEA) was less than 0.06, and both the comparative fit index (CFI) and Tucker-Lewis Index (TLI) are greater or equal to 0.95.

Table 3 presents standardized item loadings for each of the factors in the well-fitting

model. Loading values range from 0.0–1.0 and indicate the degree to which an item serves as a measure of a factor. Values closer to one signify that the item is a pure measure of that factor. Here we note that all items load significantly and highly (≈ 0.7 or more) on their corresponding factor. Further, Cronbach's alpha values for each factor met or exceeded 0.8, indicating good internal consistency within each factor, and the overall survey internal reliability was $\alpha = 0.89$. The two residual co-variance paths in the model did not exhibit high correlations, as had been the case in some of the original development studies [71]. Inspection of the modification indices revealed no suggestions that would markedly improve model fit.

Theoretically, factors that measure EL and GL should be negatively correlated. The more EL that exists, there is less working memory available for GL. Components that measure IL and GL should have a correlation around 0 indicating that the relationship between IL and GL is non-linear. Extremely low or extremely high levels of IL may lead to a lower GL score. If a learning task is too easy for a student, the explanations and instructions in the task may not contribute to actual student learning. On the other hand, if the learning task is too complex for a particular student, working memory available for GL activity may be limited.

The observed inter-factor correlations (see Table 4) support this theoretical interpretation. Overall, observed correlations were all less than 0.8, indicating a reasonable degree of discriminant validity (i.e., each factor measures a unique facet of cognitive load) [19, 93]. We observed a weak, positive correlation between intrinsic and extraneous load and a moderate, negative correlation between extraneous and germane load. Negative correlations in the table involving germane load are due to the negated language used in items 7–9 as compared to those on the rest of the survey (as previously discussed). With respect to the relationship between IL/GL, there was only low correlation ($r = -0.282$) accounting for just 7.4% of the co-variance between the two factors.

Table 5: Factor Loadings and Reliability (Lecture 2)

Factor/Item	Factor Loading	Std Err	p-Value
IL - Intrinsic Load (α = 0.86)			
Q1	0.77	0.050	< 0.001
Q2	0.95	0.037	< 0.001
Q3	0.75	0.047	< 0.001
EL - Extraneous Load (α = 0.85)			
Q4	0.90	0.033	< 0.001
Q5	0.81	0.043	< 0.001
Q6	0.72	0.051	< 0.001
GL - Germane Load (α = 0.93)			
Q7	0.85	0.033	< 0.001
Q8	0.91	0.026	< 0.001
Q9	0.89	0.032	< 0.001
Q10	0.83	0.035	< 0.001
Residual Covariance			
Q7 with Q9	-0.08*	0.156	0.620
Q9 with Q10	0.13*	0.143	0.361

*denotes a correlation rather than a loading

3.3.2 Lecture 2

Analysis of the data collected during a subsequent class lecture (N=117) also demonstrated excellent model fit ($\chi^2(30) = 39.7, p = 0.11; RMSEA = 0.053, pclose = 0.43; CFI = 0.99; TLI = 0.98$). As shown in Table 5, the standardized item loadings for this set of data followed a very similar pattern to that found with Lecture 1 responses. All items strongly loaded on their respective factors. Cronbach's alpha values demonstrated good internal consistency within the factors, and the overall reliability for the scale here was $\alpha = 0.87$. Again, none of the modification indices suggested further substantive additions to the model. Lastly, the two residual/error paths in the model (between items q7/9 and q9/10) were not significantly correlated here. Taken in combination with the low correlations between these items observed with the Lecture 1 data, they could likely be removed from the factor model entirely without negatively impacting overall model fit or its interpretation significantly. (Removing the two residual/error paths was originally suggested by [71] after analysis of Study 3; here the co-variance pairs were kept in an effort to be faithful to the original study.)

The factors correlated (Table 6) in a nearly identical pattern to what was observed with

Table 6: Factor Correlations (Lecture 2), all significant at $p \leq 0.01$

	IL	EL	GL
IL	1.0	0.403	-0.272
EL		1.0	-0.739
GL			1.0

the Lecture 1 data. Again IL and EL had a weak, positive correlation and EL and GL showed a moderate, negative correlation.

Overall, the conclusion is that the CS Cognitive Load Component Survey as adapted for introductory computer science validly replicates the original findings statistically and can reliably reproduce them across multiple lectures in a CS1 class.

3.3.3 Contributions

The contribution of this study is the creation of an introductory CS specific instrument, adapted from an existing instrument for measuring cognitive load. What follows is a hypothesized explanation of the results of the measurements collected. A thorough exploration of the differences between the two lectures would require a deeper and broader analysis than just considering cognitive load.

To analyze the perceived cognitive load levels during each of the two lectures metrics for each of the factors were determined. These metrics are computed as the average rating given all of the items within a factor and thus falls in the range 0–10. Table 7 presents the mean and standard deviation (σ) for each survey item and survey factor. Metrics from Lecture 1 are shown on the left-hand side of the table, while corresponding metrics from Lecture 2 are on the right.

The original hypothesis was that the lecture on sound processing (Lecture 2) should have posed a lower intrinsic load than the lecture on processing lists to produce HTML (Lecture 1). Recall that the IL component measures the innate connectedness or inherent difficulty of a topic (for a novice). The introduction of sound processing requires only the knowledge of sampling sounds to generate digital representations, while processing lists to

create HTML code requires knowledge of strings, lists in Python, and valid HTML code. Analytically speaking, Lecture 1's content requires holding multiple concepts in working memory at one time; more so than Lecture 2's content. The data collected in this study supports this hypothesis, and IL during Lecture 1 was statistically significantly greater than during Lecture 2 ($t(271) = 4.43, p < 0.001$).

With respect to extraneous load, it was also believed that Lecture 2 would pose lower demands on mental resources, and this was confirmed with statistical significance ($t(271) = 3.08, p = 0.002$). EL consists of instructional material that serves to distract the student from learning. Split attention between two elements, using only text explanations, and unnecessary redundant material all contribute to a higher EL score. In this case, the hypothesis that Lecture 2 would have a lower EL score because it makes use of dual input channels—both audio as well as visual inputs [33]. A known multimedia learning principle to reduce extraneous cognitive load is to use both pictures and sound rather than text only to explain a concept [33]. Even though students were most likely more familiar with HTML and web pages than with sound manipulation, the sound processing lecture produced the lower EL average. The sheer number of items being manipulated to produce the web page with Python code (Python editor window, HTML code editor window, browser window) forces split attention for the student. Even though with sound processing the editor window and sound visualization are both on screen, the student does not attempt to interpret or "read" the visualization; it is merely a placeholder and representation for the sound which is heard.

The final cognitive load factor is that of germane load, in which Lecture 1 had a statistically significantly lower score ($t(271) = -3.91, p < 0.001$). Any reduction in GL would indicate that students are utilizing too much working memory for IL and EL, thus reducing the amount of learning. According to cognitive load theory, students may have learned less from Lecture 1 than they did from Lecture 2. This could be determined through student

Table 7: Average Factor Scores by Lecture

	Lecture 1 (N=156)			Lecture 2 (N=117)		
Q	Avg	σ	Factor Avg	Avg	σ	Factor Avg
1	6.32	2.28	IL=6.33	5.18	2.38	IL=5.19
2	6.35	2.36	σ =	5.21	2.49	σ =
3	6.32	2.45	2.07	5.18	2.45	2.16
4	3.67	2.45	EL=3.59	2.74	2.18	EL=2.81
5	3.47	2.45	σ =	2.62	2.18	σ =
6	3.62	2.60	2.11	3.07	2.41	1.98
7	5.63	2.40	GL=5.54	6.63	2.07	GL=6.53
8	5.58	2.40	σ =	6.38	2.05	σ =
9	5.51	2.51	2.20	6.58	2.00	1.87
10	5.47	2.43		6.52	2.14	

performance on questions addressing the concepts covered during the two lectures. Because surveys were collected anonymously, correlation with test question performance was impossible here—and beyond the focus on the instrument itself.

3.4 Limitations

While this study adapted and initially verified the use of the CS CLCS here, there are some limitations and opportunities for further survey refinement. This study used data collected in two sections of a single introductory CS class with lectures given by the same instructor. These lectures were chosen in part because of opportunity (we wanted to ensure the same lecturer and identical lecture content) and because of the topics. Both lectures were the introduction of new topics. Wider data collection with other courses and instructors has been done by [72] in foreign language and statistics, but this study has not yet been replicated in computer science. A future step would be to repeat the study with a range of instructors across several different classes. Such a broad-based and cross-sectional approach would allow student-related, teacher-related, and subject-related factors in the item responses to be determined.

It should be noted that the original survey instrument and this adaptation were both done when Germane Load was considered to be an integral part of cognitive load. As is

evidenced by these results and others, the consensus among researchers now is that the three components –intrinsic, extraneous, and germane –are not additive to an overall sum. Instead researchers now consider that cognitive load consists of use of resources –germane resources and extraneous resources. It is now believed that instructional material can help to reduce the extraneous load and minimize the intrinsic load thus leaving the remaining working memory free for learning. Any results from the cognitive load component survey regarding the germane load should be considered preliminary. However this does not invalidate the extraneous and intrinsic load results. Recall that the measurements are based on learners' perceptions and that CFA ensures that each factor measures something unique. In [72] and these findings, the results did not show a strong ability to measure germane load. If germane load does not exist as a separate component, this would make sense.

3.5 Discussion

I hypothesize that the difficulty of learning to program lies in the cognitive load involved in the task, we must be able to measure the relative level of cognitive loads of different interventions. While we may not yet be at the stage where we can quantify in numerical terms the overall cognitive load or that of individual components of cognitive load, we can compare the cognitive load or component load between two interventions to determine which has the higher perceived load by the learners. We can then begin to put different interventions along a spectrum of cognitive load complexity. That is how the cognitive load measurement survey will be used throughout this dissertation.

CHAPTER IV

MODALITY STUDY

Cognitive load theory addresses the use of modality with respect to instructional materials in the modality effect. However this has yet to be empirically tested in the context of computer science, specifically programming. For years textbooks have presented code segments or entire programs on one page with explanations of the code on the accompanying page(s). This presentation method suffers from both split-attention and use of a single input mode, text. What if we were able to offer dual modality, text and audio, for explaining code segments within a textbook format? Would this increase learning for the students? Would the removal of split-attention allow students to learn the material quicker and increase performance?

Instructors presenting code in class make use of dual modality, while code is displayed via a shared display, the instructor explains the code. The students may concentrate on looking at the specific line of code being discussed while the instructor gives the explanation. This removes both the split-attention problem and the single modality delivery. How can we duplicate this environment without the instructor?

With the advancement of technology and creation of electronic books (eBooks) we are able to embed audio segments within the book allowing for the explanation of code segments. We call these 'audio tours' and the user interface can be seen in Figure 5. It is possible to create multiple audio explanations per code segment. In this example there is a line-by-line explanation of the code in addition to an overall or 'structural' tour of the code. As the audio plays, the corresponding line (or lines) of code are highlighted as a means of signaling [83]. Users can play the entire explanation or pause and repeat segments or skip to a segment that interests them within the audio tour.

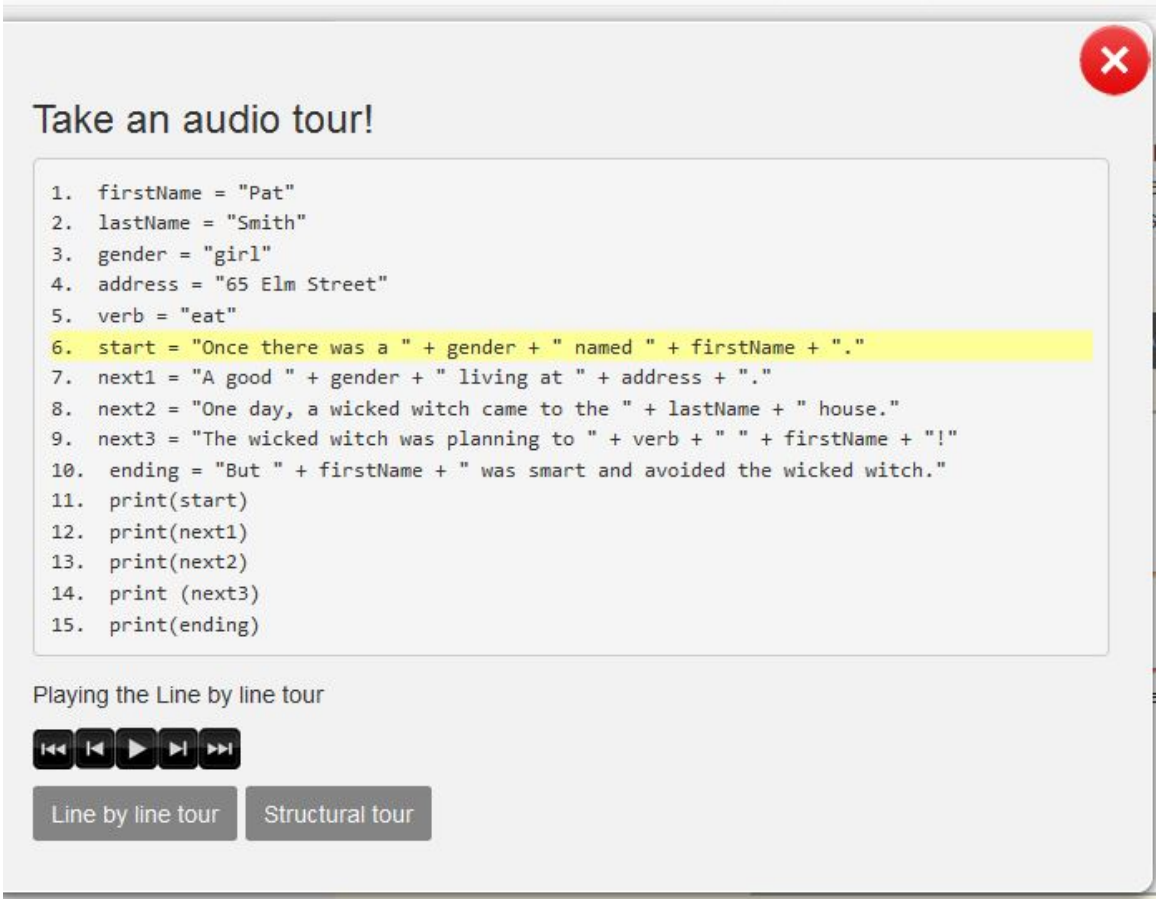

Figure 5: Audio Tour Interface

We have evidence that people see code segments more as a diagram rather than text to be read [54]. We know that expert programmers do not read a program line by line to understand it [21]. Instead they group the lines of code into 'chunks' which represent a purpose. This is similar to what chess grand masters do when they see a chess board [49]. Physics experts do something similar when examining a diagram. They classify the problem by the components within the diagram ([18, 131]). By modeling this behavior with the structural tours, we are demonstrating to novices how to interpret the code segment more like an expert.

The research in both cognitive load theory [131] and multimedia principles for learning [83] indicates presenting information using both diagrams with audio explanations yields

better learning performance than using diagrams with text explanations. In [92], the authors found that a visually presented geometry diagram, combined with aurally presented statements, enhanced learning compared to a conventional, visual-only presentation. In a split-attention situation, increasing effective working memory by using more than one modality produced a positive effect on learning. In [136] used elementary electrical engineering instructions and shows that an audio/visual diagram format was superior to purely visually based instructions. The [98] cognitive load measurement tool was used to support the suggestion that the effect is due to cognitive load factors. In [65] the authors confirmed that a dual-mode presentation of instructional information is a viable alternative to physical integration of all written materials (eliminating split-attention) within an elementary electrical engineering domain. In addition, Mayer [84] presents evidence of several studies done in the multimedia medium with animated videos and spoken explanations with findings indicated that the spoken explanation was only effective when it was done simultaneously rather than sequentially with the visually presented information.

Specifically [65] [p.369] state the following:

> Using cognitive load reduction as an overarching principle, the current experiments suggest that when dealing with split-source diagrams and text (1) materials should not be presented in both auditory and written form; (2) textual materials should not be presented in both auditory and written form; (3) if textual materials must be presented in written form, search for diagrammatic referents should be reduced by using appropriate markers or guides such as color-coding. These principles, based on cognitive load theory, can provide some guidance in the design of multi-media instruction.

This chapter reports on a study designed to empirically test the modality effect in computer science. All of the above criteria were applicable and used in the design of the experiment. In essence this study is a replication of [92, 136, 65] within the computer programming domain.

Specifically this study was examined to answer **RQ1:Does altering the modality (text, oral, both) of code explanations improve student learning as measured by retention and transfer questions?**

4.1 Study Method

A series of three videos were designed, each with the purpose of explaining a single segment of code. Three different introductory programming topics were addressed: assignment with mathematical operators, nested selection (if) statements, and finding an element within a collection (using a for loop). In addition, an appropriate context or real life scenario was derived to motivate the problem. In video 1, the problem is summing lines of an invoice, calculating the tax due and the final total for the invoice. In the second video, determining whether or not a donor and recipient have compatible blood types is explored. In the final video, finding the next possible movie time from a list of movie times and knowing the current movie time is presented.

4.1.1 Instructional Materials

See Figure 6 for the exact code for each video. Within each video the problem was presented followed by an explanation of the code. Each explanation presented the overall solution outline followed by an explanation of each line of code, much like an instructor would do in class. Each video then concluded with a trace of one or more traces of execution of the code with sample values.

After each video the participant was asked a series of questions. The first question always asked the purpose of the code segment. This was followed with one or more recall questions concerning the purpose of variables or interpretation of a given line of code. One or more application questions were presented, asking the participant to predict the output for a segment of code from the original example. The final questions were transfer questions. All transfer questions were taken in their original form or adapted from [1]. The final question allowed the participant to indicate if there were any technical issues with the

```
1)  qty = 15
2)  price = 15.67
3)  amt = qty * price
4)  total = amt
5)  qty = 250
6)  price = 0.32
7)  amt = qty * price
8)  total = total + amt
9)  tax = total * 0.08
10) total = total + tax
```

Video 1

```
1)  current_time = "14:00"                          #2:00 is 1400 in military time
2)  found = False                                   # indicates finding a time
3)  movie_times = ["13:00", "15:30", "18:00",       # list of movie times
4)                 "14:30", "17:00", "19:30",
5)                    "13:30", "16:00", "18:30" ]
6)  for element in movie_times:                     # for each element in movie_times list
7)      if current_time < element:
8)              if found == False:    # if this is first value found < current time, save it
9)                  time = element
10)                 found = True   # we've found one time, so set found to true
11)             else:                             # if we have previously found a time,
12)                 if element < time:        # check if this time is sooner
13)                         time = element
14) if found == False:                            # this means we never found a time
15)     print "You missed all the movies for today."
16) else:
17)     print time
```

Video 3

```
1)  def match  (recipientType, recipientRh, donorType, donorRh):
2)          compatible = False                  #assume no match, only change if there is a match
3)          if donorType == "O":                #match types first, then do Rh
4)              compatible = True
5)          elif donorType == "A":
6)              if recipientType == "A" or recipientType == "AB":
7)                  compatible = True
8)          elif donorType == "B":
9)              if recipientType == "B" or recipientType == "AB":
10)                 compatible = True
11)         else:                   #donorType is "AB"
12)             if recipientType == "AB":
13)                 compatible = True
14) #now consider Rh;
15)     if compatible == True:              #only have to match up those with blood types that match
16)         if donorRh == 1:                #if donorRh is positive, then recipientRh must be positive
17)             if recipientRh == 1:
18)                 compatible = True
19)             else:
20)                 compatible = False
21)     return compatible
22)
23) recipientType = "AB"
24) recipientRh = 1
25) donorType = "O"
26) donorRh = 0
27) if match(recipientType, recipientRh, donorType, donorRh)==True:
28)     print "it's a match"
```

Video 2

Figure 6: Modality Code Examples

video. There were a total of eight questions after each video.

The first example is straight forward using mathematical operators and the assignment statement to compute the total price from quantities and prices in an invoice. After the video there were four recall questions concerning the purpose of the code, number of invoice lines processed, and the purpose of variables. There was one application question concerning the ability of a variable to appear on both sides of an assignment type. This was followed by two transfer questions involving assignment statements only. The final question asked about technical difficulties.

The second example involves nested selection statements and determining if a donor's and recipient's blood types are compatible, including the Rh factor. Participants were shown the problem definition along with a chart indicating blood table compatibility (Figure 7). Two examples were described on how to read and interpret the table. The code was then explained line by line, followed with two examples tracing through the code, one for a compatibility match and one for an incompatible match. Recall questions asked were the purpose of the code, possible values for a boolean variable, the name of the function, and what was the second thing checked for blood compatibility. The one application question involved tracing of a portion of the nested selection statements. Two transfer questions were asked, both with nested selections statements. The final question asked about technical difficulties.

This second example was written to include a "main" program along with a function call and function definition. This was done purposefully to allow for the easy changing of the values of the variables for compatibility testing.

The third video was finding the next possible movie time from a list of non-sequential movie times and involved a loop. The participant was given the problem definition, an outline of the solution approach followed by a line by line explanation of the code. The video concluded by tracing through two executions with sample values. The recall questions for this video were the purpose of the code, understanding the solution, and representation of

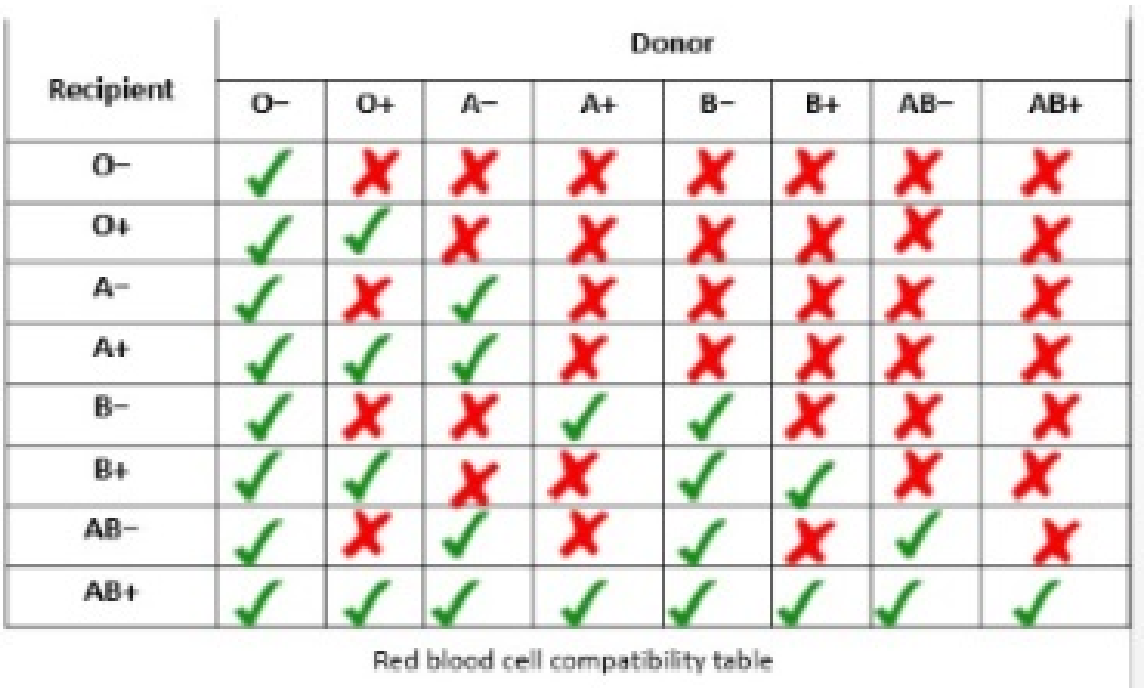

Figure 7: Red Blood Cell Compatibility Table

the data. The application question asked the user to determine what would happen if all the movie times had already passed for the day. There were three transfer questions and the final question on technical difficulties. All loops in the example and questions used the for loop format.

It should be noted that the videos were designed with the purpose of minimizing cognitive load. During the line-by-line explanation of the code, there was signaling indicating which line was being discussed (Figure 8). When examples were being traced, the variable values and results of comparisons were integrated into the code diagram (Figure 9). The second example given in each video allowed for a pause for participants to attempt to trace through the code on their own before the solution was presented.

A script for the code explanation was created for each problem. For each video, three different versions were created–one with an audio only code explanation, one with a text only explanation, and one that combined both the text explanation and the audio explanation. See Figure 10 for what the study participant might see. This shows how the text

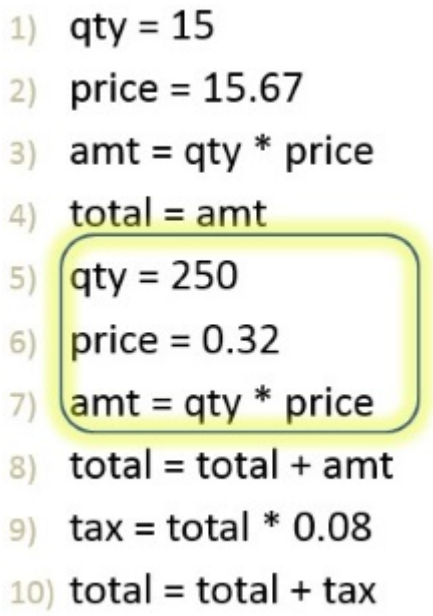

Figure 8: Signaling Example

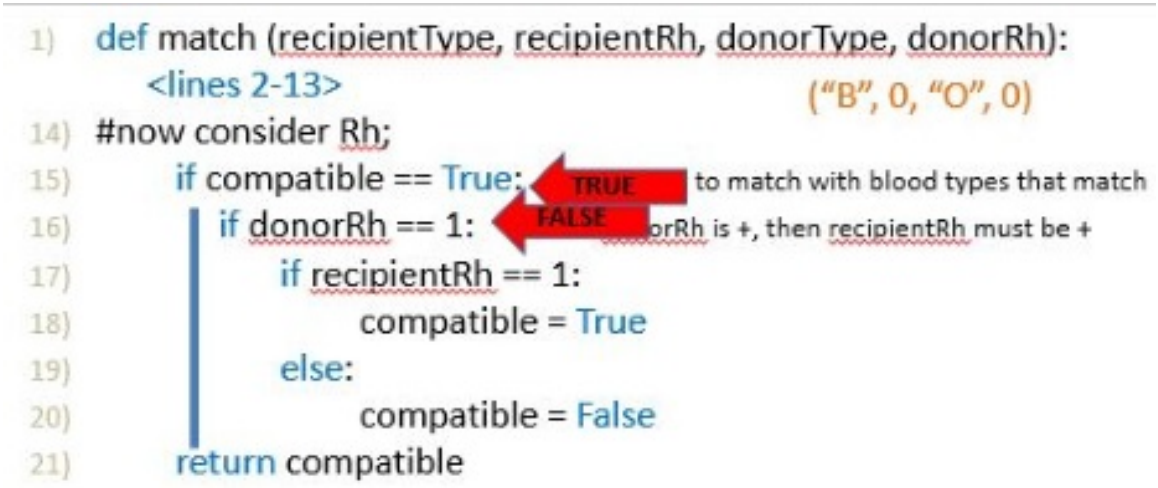

Figure 9: Code Tracing Example

explanation was presented to the user. The line of code currently being explained was highlighted in all three treatments. At the conclusion of each video a summary page was presented with all of the relevant information presented.

The time, with one exception, was controlled for within each treatment. The time spent on each screen was constant for all three versions. Using the reading time of an average 17

year old it was determined how long it took to read the current text on the screen (plus a slight delay) and the audio was controlled to match that time. The one exception was when the participant was asked to trace through the code for the second example in each video. The instructions asked the user to pause the video while the code was showing and walk through the example. They were asked to restart the video when they knew the expected output.

4.1.2 Participants

Participants were recruited from introductory programming courses or breadth-first (CS0) courses at multiple universities in the southeast United States. Having read and given consent, participants were given a pre-test in order to eliminate those that had too much programming knowledge. Based upon the day of their birth, they were assigned to one of the three study conditions (audio only, text only, or both audio and text). After viewing each video they were asked to complete the CS cognitive load questionnaire described in the previous chapter, followed by a series of questions designed to determine how much information they recalled, how much they could apply, and questions designed to test transfer of knowledge. At the conclusion of watching the videos they were asked a series of demographic questions. The demographic questions were moved to the end to prevent stereotype threat [124, 125]. The average age of participants whose data was analyzed was 21.04, with a minimum of 18 and a maximum of 42. The median age was 19. In terms of their native language, 39 of the participants spoke English, 6 spoke Dutch, 2 spoke Korean, and 8 spoke some other language.

4.2 Data Analysis

Data was collected from August 13, 2014 to November 29, 2014. The results were downloaded from SurveyMonkey and analyzed. A total of 141 participants agreed to the consent. Those participants with no answers (all blanks) were eliminated, leaving 99 responses. The pre-test answers were scored for correctness, and those with a score of 67% or better (6 out

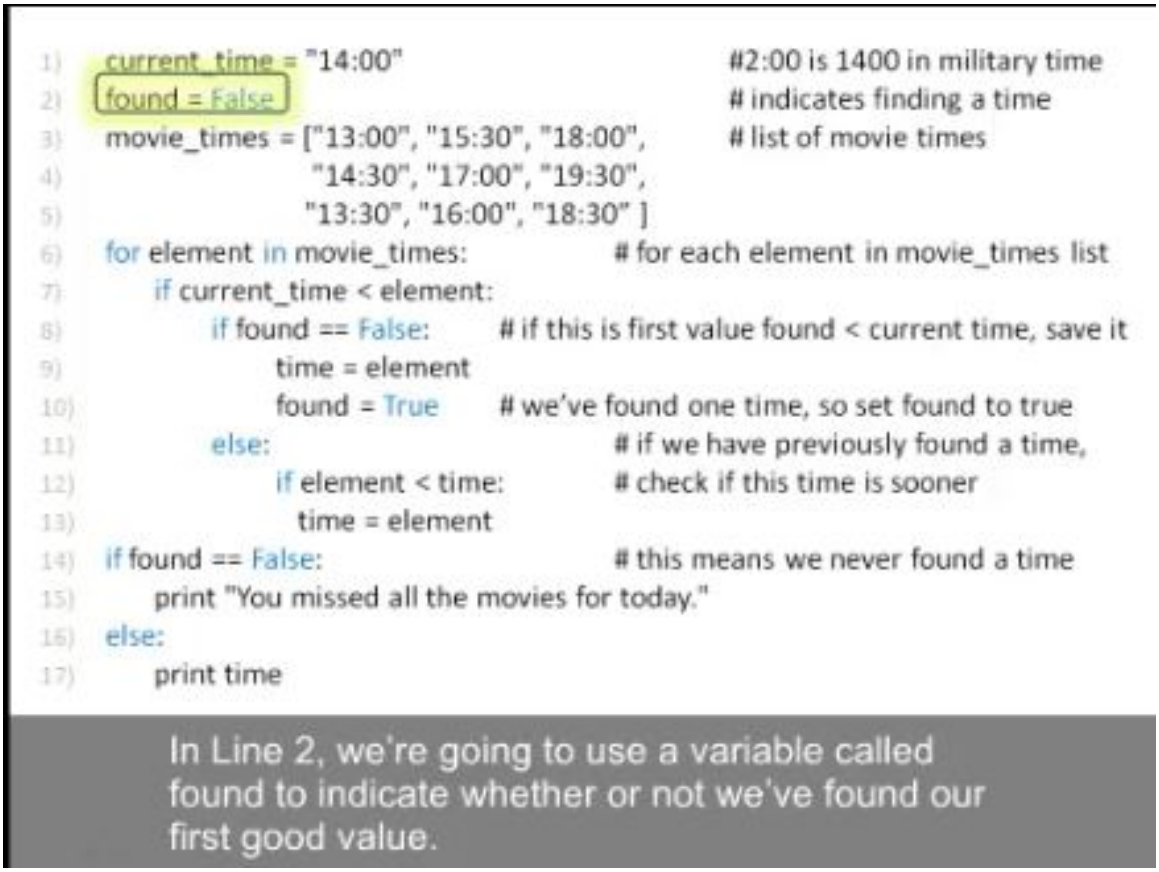

Figure 10: Modality Study Example

of 9 possible points) were eliminated for having too much knowledge. This left 88 respondents. Those with no answer to the day of birth question (thus no treatment choice) were eliminated, leaving 77 respondents. These were then disaggregated by treatment (Table 8).

Table 8: Participants by Treatment

Treatment	N	Removed Due to No Answers	Final N
Audio	27	3	24
Text	31	5	26
Both	19	8	11

Many participants did not view all three videos (Figure 11). Participants were given the option to quit the study after answering questions for each video. In pilot studies for this experiment, the most common complaint was the overall length of the study and exiting early from the study. In the pilot version this eliminated the possibility for collecting any

demographic data since it was originally positioned after the last video. As a solution to this problem, the incentive for the participants was changed to one raffle entry for each video and set of questions completed, and an extra raffle entry if all three videos were completed. In addition, if the participant chose to quit after answering questions for a video they were directed immediately to the demographic questions so that information could be collected from all participants.

For the text only group, 26 participants watched the first video while only 10 continued to the second video (a 62% attrition rate) and 11 continued to final video. One participant of the 11 indicated that the second video did not play within their browser. For the audio only group, 24 participants watched the initial video, but only 17 watched the second video (a 29% attrition rate) and 16 watched the final video. For the audio and text group, only 11 participants actually completely watched the initial video and answered questions, 8 watched the second video (a 27% attrition rate) and 7 watched the final video. Generally, if the participant continued on to the second video they watched all three videos. It is interesting to note that while using day of the month of their birthday to randomly assign a participant to a treatment, it did not end up random. For the text group, 36 participants were assigned, 30 were assigned to the audio group, but only 22 were assigned to the "both" group. Thus for the text only group, 62% of the participants assigned to the treatment went on to watch the first video and have data analyzed. For the audio only group, 80% of the participants completed the first video and had data analyzed. However for the "both" treatment, only 50% of the participants assigned to the treatment had their data analyzed. This may be because these participants found having both video and audio explanations confusing or cognitive overload and chose to end their participation in the experiment early.

The average and standard deviation were calculated for the cognitive load components for each video per treatment (Table 9).

We can graph these results for easy comparison. See Figures 12, 13, and 14.

In all three groups the germane load (GL) was perceived as the highest component

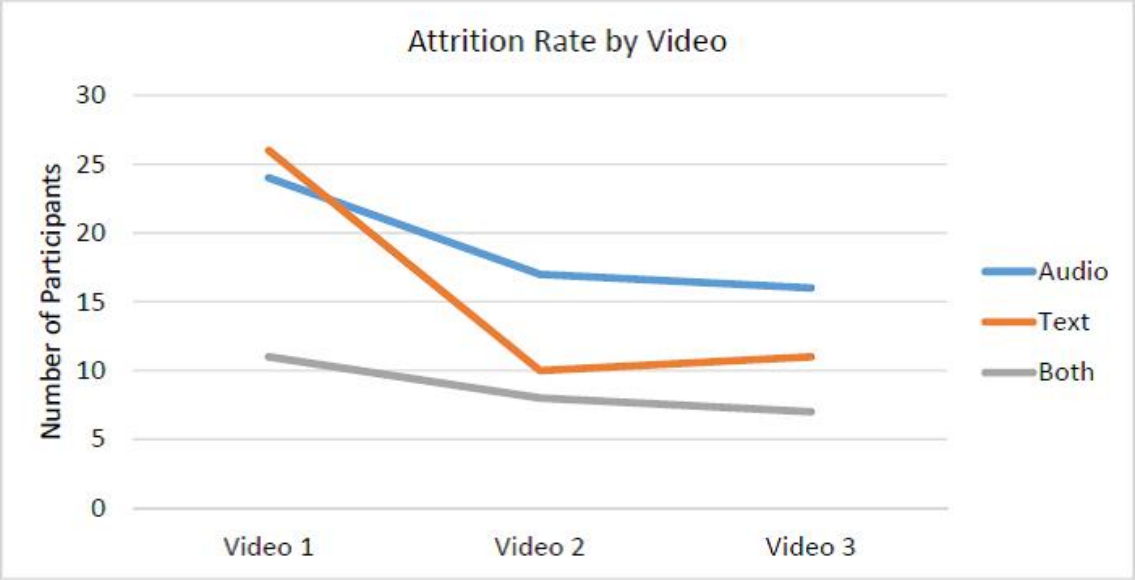

Figure 11: Participant Attrition by Video

Table 9: Cognitive Load Components by Video / Treatment

Treatment		**Video 1**	**Video 2**	**Video 3**
Audio	N	24	17	16
	IL avg (stddev)	3.35 (3.12)	4.69 (3.50)	3.92 (2.70)
	EL avg (stddev)	1.75 (2.05)	1.85 (2.38)	2.23 (2.78)
	GL avg (stddev)	6.10 (2.77)	5.18 (3.36)	6.09 (2.93)
Text	N	26	10	11
	IL avg (stddev)	3.31 (3.05)	4.27 (2.32)	4.27 (2.54)
	EL avg (stddev)	2.94 (2.88)	3.17 (2.55)	2.85 (2.44)
	GL avg (stddev)	6.04 (2.88)	5.5 (2.36)	5.34 (2.56)
Both	N	11	8	7
	IL avg (stddev)	2.55 (3.08)	3.92 (3.02)	2.62 (2.27)
	EL avg (stddev)	1.70 (2.04)	1.96 (2.06)	2.10 (2.76)
	GL avg (stddev)	6.14 (2.51)	6.34 (2.06)	

and the extraneous load (EL) was perceived as the lowest component. In addition video 2 consistently had the highest intrinsic load (IL) measure.

We can also look at the post-test results to determine what learning occurred. The post-test questions for each video were scored for correctness. For multi-select questions, the number of incorrect choices were subtracted from the number of correct choices to get a final score. The results for each video can be seen in Table 10.

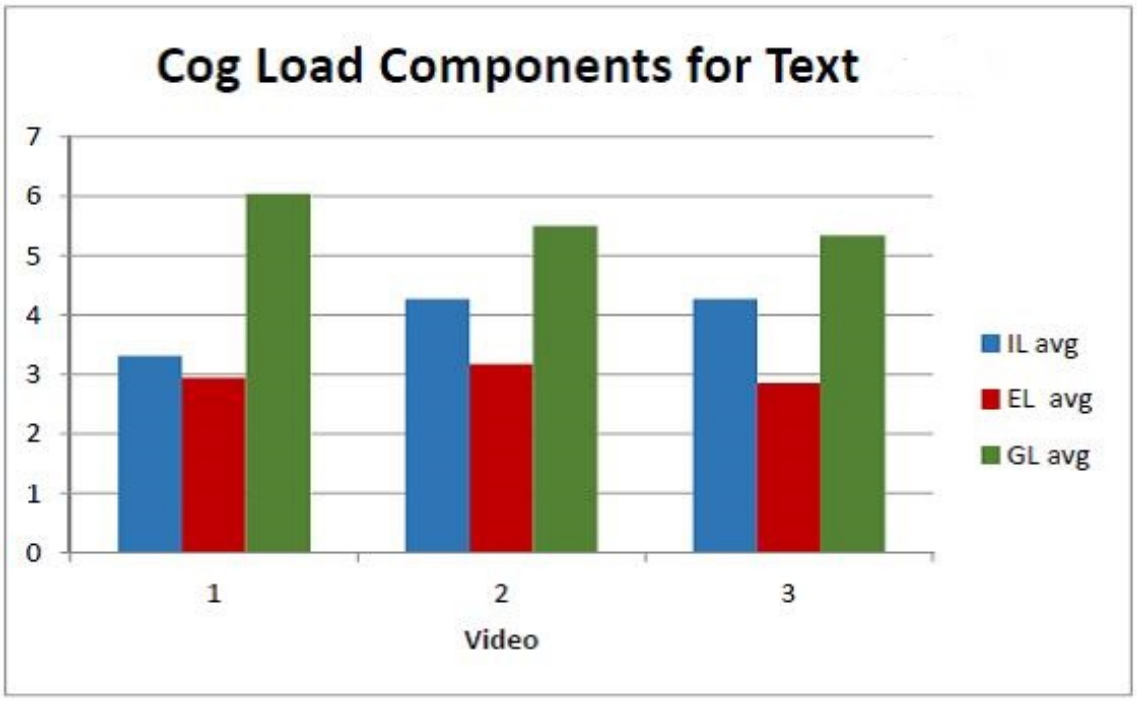

Figure 12: Cognitive Load Components for Text Treatment

We can break down the results by question type. See Figures 15, 16, and 17 for participant performance by treatment.

A statistical analysis was done to determine if any correlations existed between treatment and participant performance. All results were statistically insignificant. There was no main effect for treatment, F (2, 52) = 0.178, MSE = 1.145, p = .837. A statistical analysis was done to determine if any correlations existed between the cognitive load factors and treatment. All results were statistically insignificant (IL, p = .375, EL, p =.715, GL, p = .628).

4.3 Results and Contributions

A comparison of overall participant performance by video was completed. The results can be seen in Figures 18, 19, and 20.

The results from this study do not match what was found in the original studies, and

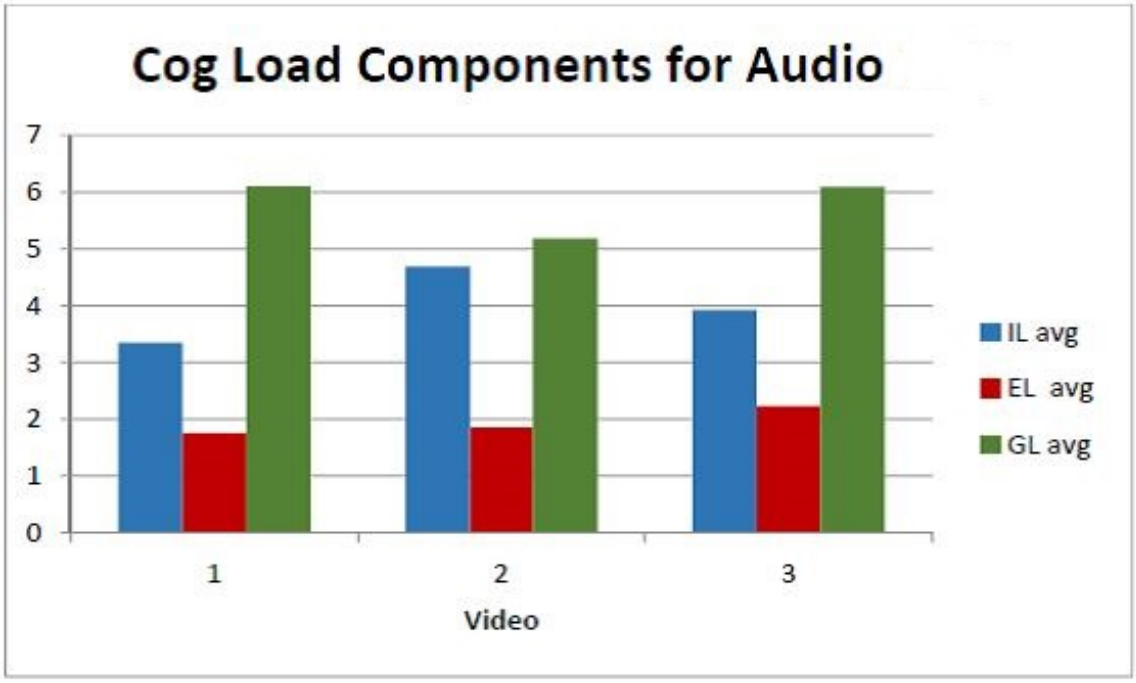

Figure 13: Cognitive Load Components for Audio Treatment

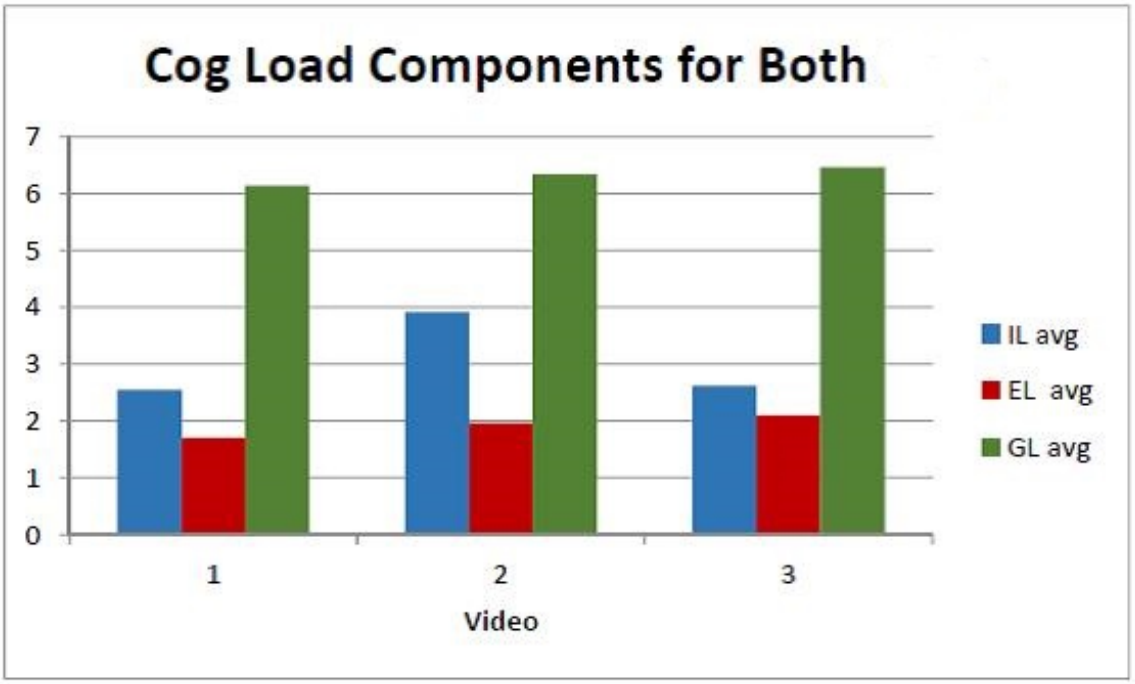

Figure 14: Cognitive Load Components for Both Treatment

Table 10: Post Test Results of Modality Study

Treatment		Video 1	Video 2	Video 3
Audio	Purpose (% correct)	3.13 (54)	3.06 (65)	3.29 (56)
	first question (% correct)	46	47	56
	second question	83	47	19
	third question	50	82	75
	fourth question	42	47	44
	fifth question	33	6	19
	sixth question	17	41	19
Text	Purpose	2.57 (42)	3.4 (80)	2.91 (64)
	first question (% correct)	23	60	73
	second question	46	20	9
	third question	46	100	73
	fourth question	23	90	55
	fifth question	38	40	18
	sixth question	46	60	27
Both	Purpose	2.46 (27)	3.5 (88)	2.29 (29)
	first question (% correct)	18	50	57
	second question	64	75	0
	third question	45	75	43
	fourth question	45	50	43
	fifth question	36	25	29
	sixth question	36	38	29

what was expected in this study. In the auditory only group, the code explanations were given aurally only along with color coding signaling. This was the group that was expected to have the best performance, especially with recall questions. If the modality principle was to hold in learning computer science, the audio treatment participants should have scored statistically significantly higher than the other two groups. The group that received both written text and an auditory explanation was expected to perform the worst on the learning performance tasks. The "both" group should have scored the worst. These predictions should have held for at least the first and most simplest of the videos, but it did not.

In answering **RQ1**, we find no evidence in support of either hypotheses: H1A: Students receiving oral explanations will demonstrate better retention. H1B: Students receiving both oral and text explanations will demonstrate the worst retention.

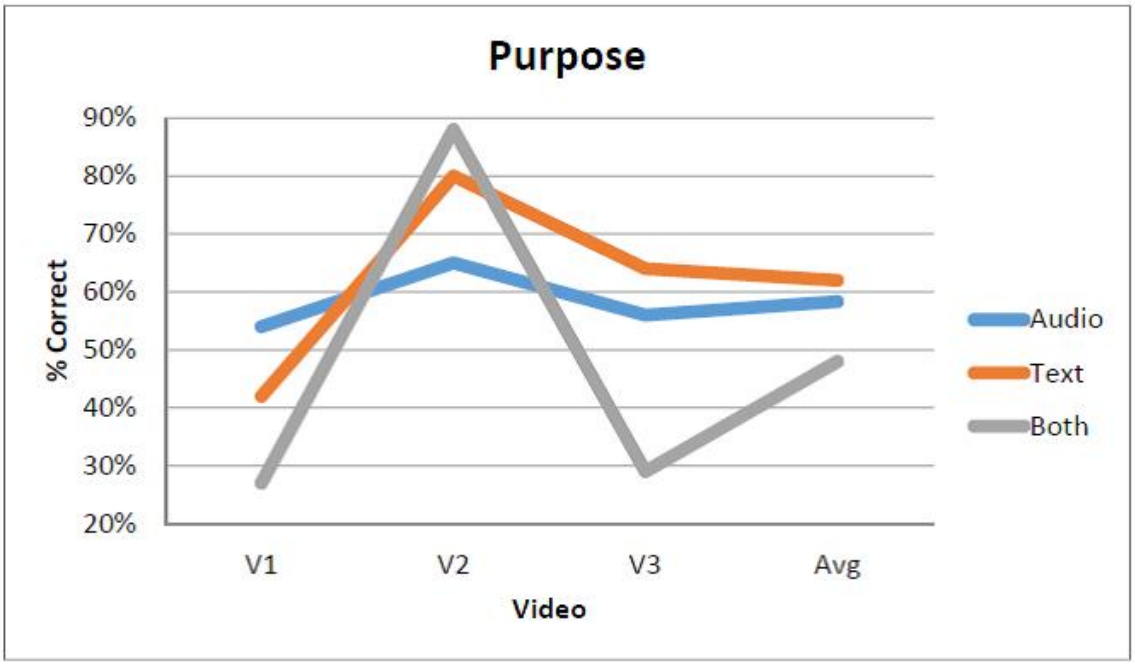

Figure 15: Performance on Purpose Question by Treatment

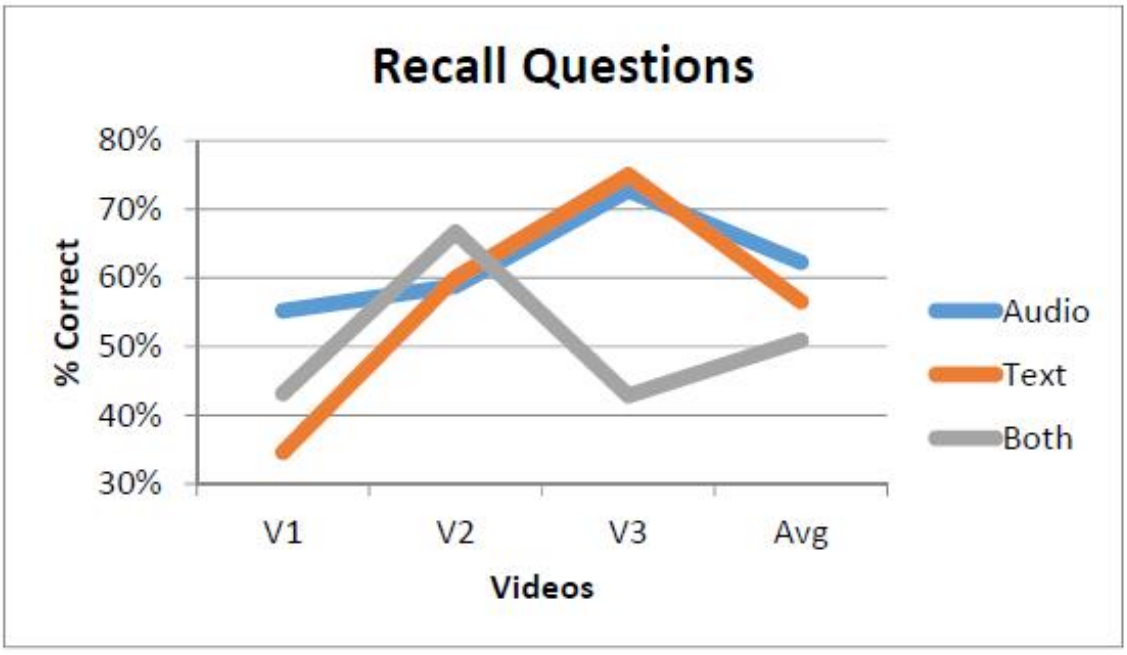

Figure 16: Performance on Recall Questions by Treatment

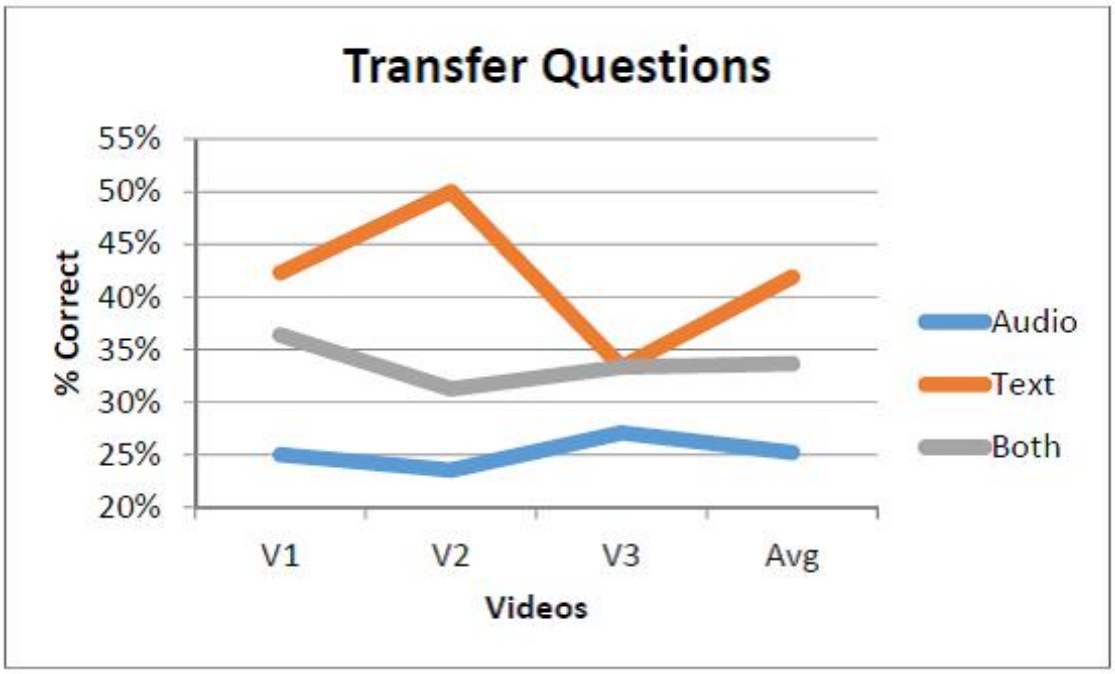

Figure 17: Performance on Transfer Questions by Treatment

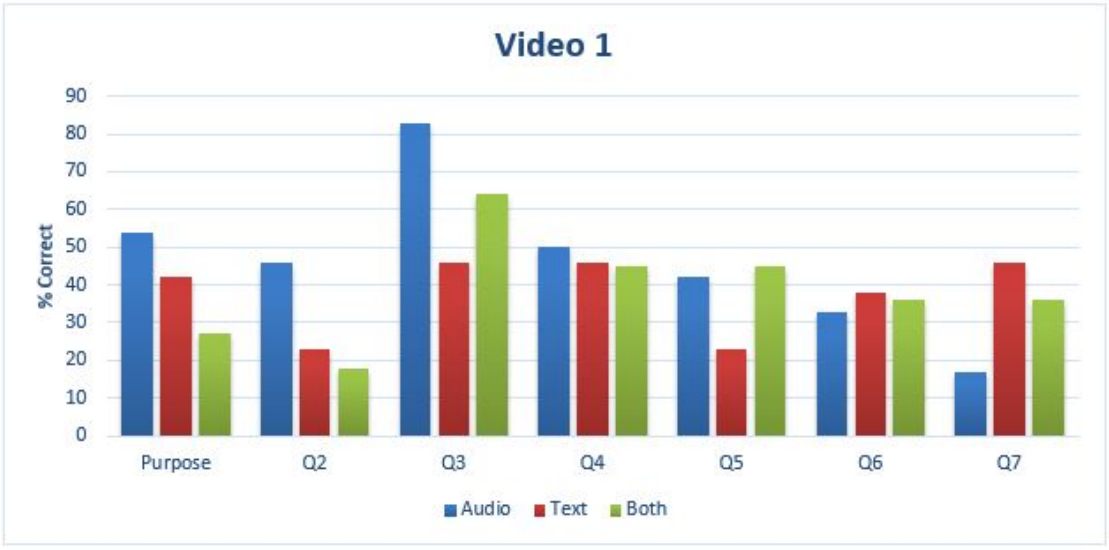

Figure 18: Question Performance on Video 1 by Treatment

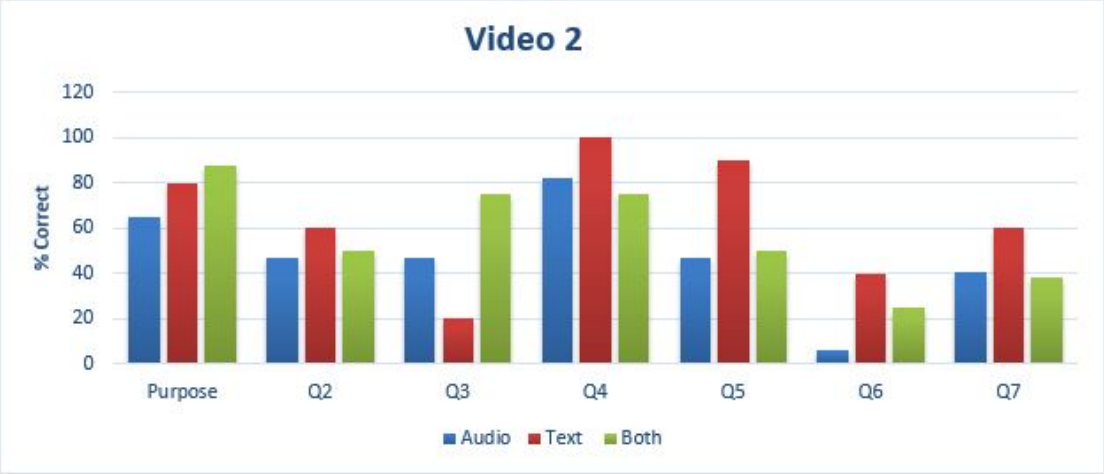

Figure 19: Question Performance on Video 2 by Treatment

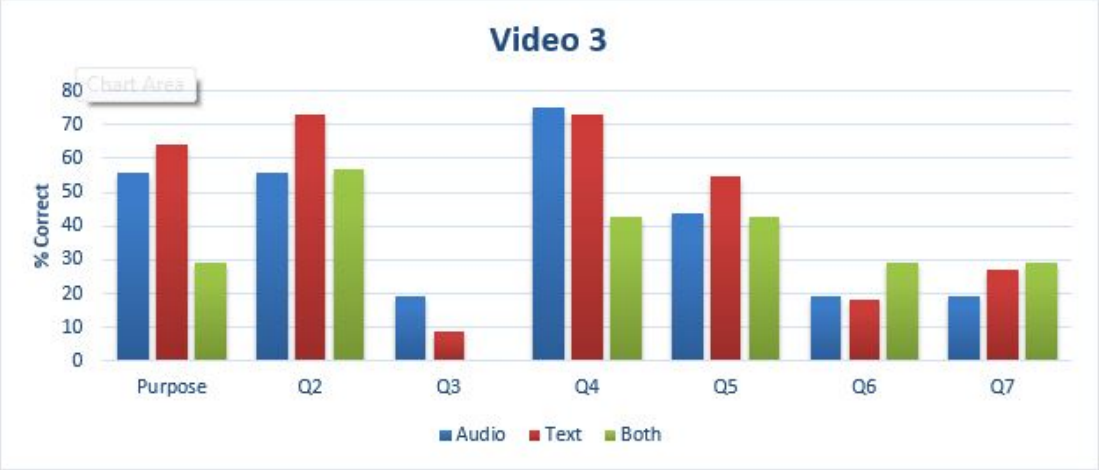

Figure 20: Question Performance on Video 3 by Treatment

4.4 Limitations

Because the anticipated results were not obtained, we must examine if it is due to the fact that the principle does not hold within the programming domain or if there are other explanations. It is possible these results were obtained due to low and uneven participant rates. It may be possible to obtain the predicted results with more, and more evenly distributed participants across the treatment conditions.

Kalyuga [61] posits two conditions when the modality effect may not be found: (1) when equivalent auditory and visual explanations are presented concurrently, and (2) when the instructional format is not matched to learner experience. While care was taken to ensure that the material was synchronized, there were two comments indicating that the participants' native language was not English and they were unable to keep up with the speed of the information presented. One of the participants was in the audio only group, the other was in the text only group.

It is also possible that the anticipated results were not obtained because of the transient effect [69, 153]. The participants in the study were novices or had minimal computing knowledge (only those that failed the pre-test had data analyzed). It is possible that even the easiest and shortest of the videos overloaded their cognitive mental processing abilities. Knowing that we can only hold information in memory for no longer than 20 seconds [36], the videos may have been too long for the participants to comprehend and understand all the material asked of them. Video 1 was 5 minutes long; the second video was almost 23 minutes long and video 3 was 12 minutes long. However in [69] the modality effect occurred when the videos were *longer* (going from 605 seconds to 867 seconds) but fewer words (668 words to 576 words). In Leahy and Sweller's second experiment the explanation was simplified into smaller segments and less complex sentences. For my experiment, video 1 contained 528 words in 14 slides, video 2 contained 3167 words in 63 slides, and video 3 contained 1901 words in 33 slides. While video 2 and 3 were both lengthy in both time and words, video 1 was completely in line with those used in previous studies which reported these values.

Each video was designed to be a replacement for an in class lecture for that content topic. It is possible that each example was over-reaching in its goal. Instead of trying to instruct about everything on a specific topic in a single video it may be better to concentrate on a small piece of a topic per video. In other words, rather than covering every possible aspect of a selection statement, it may be better to have a shorter video that only explains

what happens when a conditional expression evaluates to true in a selection statement. Then have a separate video for when it evaluates to false. Then another video for a nested selection statement. This would make each video much shorter and allow for progressive building of the information.

It is also possible that no modality effect was found due to the complexity of the material. Kalyuga [64] provides an excellent overview of the modality effect and studies which both find and do not find the modality effect in an effort to identify factors influencing its existence. The paper also gives instructional recommendations based on his findings. The one consistent recommendation is to not use spoken explanations for any material which is highly complex. In this study it could be that learning programming is so complex that no modality effect was found. It is interesting to note that no reverse modality effect was found either. If the material was inherently complex then the text only group should have performed better, which also did not occur.

4.5 Future Studies

There are a few different possible next steps to attempt to determine when, if, and how the modality principle applies to introductory programming.

- Think Aloud Study: Recruit a minimum of three participants for each treatment and have them watch the videos. Instead of controlling for time, the researcher could pause the video at different times and ask the participant to think aloud about what they have learned, what they can remember, and probe about possible transient effects. As the participants are answering the post video questions, they would be prompted to think aloud their thought process on how they are arriving at their answers. By capturing information during the learning and assessment process it may be able to more accurately determine what is occurring to explain the unanticipated results.

- Restructure the Videos: It may be possible to rework one of the three videos into

separate and more distinct content topics as described above. Another set of of participants could be recruited to determine if shortening the videos and scaling back the content makes a difference in the overall results.

- Use the eBook Audio Tours: Within the electronic book (eBook) we are developing within our lab, we have included Audio Tours (see Figure 5) to explain code examples. Within the eBook the shortest, longest and a middle length video could be determined. In order to determine a video length where the transient effect begins to occur within the programming domain, the Audio Tours could be reworked to present either an audio only or text only explanation (the "both" case would be eliminated). After each Audio Tour participants would be asked recall and transfer questions.

At this time it is impossible to state that the modality effect does not hold within learning programming. This initial evidence indicates that it may not be as simple as replicating existing studies. Instead, more information is needed on several topics. First, the transient effect within programming should be studied to determine at what point the learner begins to lose information. Exactly how much content can be explained in a single audio explanation should be evaluated as well. It may be possible to cover only the simplistic, straight forward case for a control structure without deviating to the exceptions, or even including a trace of code. More studies should be conducted.

CHAPTER V

USING SUBGOAL LABELS TO IMPROVE LEARNING PROGRAMMING

Learning to program is difficult. Students often perform poorly in introductory programming courses [16]. It may be that students have a difficult time because the instructional material used to teach programming may actually overload students' cognitive abilities. Better designed materials may enhance learning by reducing unnecessary load. Subgoal labels have been shown to be effective at reducing the cognitive load during problem solving in both mathematics and science [25, 30]. Until now, subgoal labels have been given to students to learn passively. We report on a study to determine if giving learners subgoal labels is more or less effective than asking learners to generate subgoal labels within an introductory CS programming task. The answers are mixed and depend on other features of the instructional materials. We found that student performance gains did not replicate as expected in the introductory CS task for those who were given subgoal labels. Computer science may require different kinds of problem-solving or may generate different cognitive demands than mathematics or science. [1]

As educators, we want to simplify the learning process to provide the maximum results. As researchers, we want find empirical evidence for what exactly it means to simplify the learning process. One proven method for enhancing learning is to reduce unnecessary cognitive load on the student while they are trying to learn to solve problems [106]. There are several ways to reduce cognitive load while learning to program, including using worked examples [72] instead of writing code from scratch.

[1] This work was done with the assistance of Lauren Margulieux who helped with study design, collection of data, and statistical analysis results.

Worked examples typically include a problem statement along with a step-by-step procedure for how to solve the problem. Worked examples are most effective when used in worked example-practice pairs [6]. In these pairs, students study a worked example solution and immediately practice by solving a similar problem.

Segmenting worked examples and including subgoal labels have also been shown to be effective in improving learning [6]. Segmenting includes separating portions of the worked example to isolate each advancement in the problem solution process [123]. Each advancement in the problem solution can have several individual steps. Subgoal labels are names given to a set of steps in the solution process allowing the user to "chunk" the information to ease learning [25].

Using worked examples as a means to reduce cognitive load is effective because it constrains the learner's search space. In solving an open-ended problem, the learner must search all of their working and long-term memory for any schema related to the problem at hand. This becomes an exhaustive search and is only efficient when the learner recognizes the problem category and has automated recall of the solution process. This is certainly not true for the novice within the domain who is being presented the problem and the problem category for the first few times. When studying the worked example, the learner has only to determine how the example goes from one step to the next –a very reduced search space which is a means-end search (i.e., they know the end result and must only find a path to get to that one end). This reduces the amount of cognitive processing being done by the learner [131].

When using subgoals within a worked example, it gives the student a framework for learning the solution process. It allows them to organize the solution process and attach it to prior knowledge. By providing the structure for the schema it may allow faster automation. Using subgoals also encourages "chunking" of pieces of the solution, providing another level of abstraction for the learner. Once the student sees the subgoal of "Declare and initialize variables" and several instantiations of how that can be accomplished they

can begin to chunk that type of code together and automate its recall without having to memorize the individual code lines. Giving the learner pre-defined labels for chunks may reduce the cognitive load associated with learning the solution process.

While using worked examples and subgoals to reduce cognitive have been empirically tested in math and science disciplines, we have been the first to test these with computer science learning [77]. Margulieux et al. [77] demonstrated learning benefits for subgoal labels with a drag-and-drop programming language. This paper reports on a study undertaken to empirically determine the effectiveness of worked examples and subgoal labels within introductory computer science using a more traditional textual language. Some of the findings confirm the results from other disciplines while some were unexpected.

Specifically, instructional material was created to teach introductory programming students about the process of using and writing a while loop to solve programming problems. There were three treatment conditions: (1) no subgoal labels provided, (2) subgoal labels given, and (3) subgoal labels generated, in which students were asked to generate their own labels for groups of solution statements.

Within each treatment group, participants were randomly assigned to either an isomorphic or contextual transfer group. In the isomorphic transfer group, the problem to be solved in the worked example-practice problem pair was identical to the worked example in both procedural steps and cover story (i.e., context). The only thing changed was the actual values of the numbers to be calculated. In the contextual transfer group, the problem to be solved in the worked example-practice problem pair involved the same procedural steps but the cover story and numeric values changed. As an example, the first worked example was calculating the average tip from a list of tips. For the isomorphic group, only the list of tip values changed from the worked example to the practice problem. For the contextual shift group, the practice problem involved calculating the average amount of rainfall collected from a rain gauge. The steps involved in the solution are identical in both cases, but the story motivating the solution changes.

Participants' learning was measured with performance on novel problem solving tasks and a post-test. Problem solving tasks during the assessment were different from practice problems solved during the learning acquisition phase.

We used a Parsons problem assessment in additional to the open coding assessments to determine if the performance gains found with subgoal labels still apply with a different type of assessment and if the relative performance speed replicated previous studies.

The contextual transfer was intended to promote deep learning instead of superficial learning as the contextual transfer groups had to do non-superficial transfer during learning [17, 42, 99]. In the design of the experiment we wanted to determine if adding contextual transfer affected the learning gains any differently than those who learned without contextual transfer while utilizing subgoals. In theory, studying similar worked examples (isomorphic groups) may lead to rote memorization of the labels without having to explain how the labels are actually implemented within the code. In the contextual transfer groups, having to resolve and self-explain how the labels are implemented in more than one context may lead to deeper learning. Using this logic, groups seeing contextual transfer problems should perform better than those who only see isomorphic problems.

We anticipated that transfer would not necessarily improve performance on the Parsons problem assessment. This is because in a Parsons problem the learners do not have to figure out how to apply a conceptual understanding of the procedure to a specific problem. Instead, since all the lines of code in the Parsons problem are provided for them they need only rearrange or number them into the appropriate order. This again would be more of a means-end search process than an open-ended search for the learner.

5.1 Study Method

5.1.1 Purpose

Participants in introductory programming classes were given instructional material designed to teach them to solve programming problems using while loops. This common introductory programming task requires only minimal prior programming knowledge (arithmetic operations and Boolean expressions) to complete at a basic level. The study was conducted before students had formally learned about while loops in their courses. Loops were also chosen as they have proven to be difficult for students to learn [70]. Participants were recruited from 7 different introductory programming courses at two technical universities in the Southeast United States. At one institution the study was conducted over a two week period; at the other institution the study was done over a month period. Because the courses teach different programming languages (see Table 11), pseudo-code was used in the task to make it independent from any one programming language.

Pseudo-code is relatively easy for programmers to understand regardless of the programming languages that they know [134]. The study was conducted in either a closed lab setting with up to 30 computers in a single room (one institution) or completely through email and over the internet (second institution). Students were given an explanation of the study. They worked independently. The sessions typically lasted between 1 and 2 hours, depending on the rate at which participants completed the tasks.

Table 11: Classes Participating in Study

Programming Language	**Majors**
C++ or MATLAB	Engineering
C#	Game Development
Java or Python	Computer Science, Information Technology, Software Engineering, Non-Majors (mostly physics and math)

5.1.2 Instructional Materials

To learn the procedure for using while loops to solve programming problems, participants were given three worked examples and three practice problems. See Appendix A as an example of the first worked example problem for the subgoal given group. The worked examples and practice problems were interleaved so that after studying the first worked example, participants solved the first practice problem before moving on to the second worked example. The worked examples came in three formats, which varied between participants. The first format was not subgoal oriented, meaning that steps of the examples did not provide any information about the underlying subgoals of the procedure. The second format grouped steps of the example by subgoal and provided meaningful subgoal labels for each group as is typical in subgoal label research (e.g.,[71]). The third format grouped steps of the example by subgoal and provided a spot for participants to write generated subgoal labels for each group. Each of the groups was numbered as "label 1," "label 2," etc., and groups that represented the same subgoal had the same number; therefore, groups that represented subgoal 1 were numbered as "label 1" regardless of where in the example they appeared (see Figure 21). Participants were told that each of the worked examples would have the same subgoals, and they were encouraged to update and improve upon their generated labels as they learned more.

No labels	Given Labels	

sum = 0
lcv = 1
 	
WHILE lcv <= 100 DO
 	
sum = sum + lcv
 	
lcv = lcv + 1
 	
ENDWHILE
	Initialize Variables
	sum = 0
	lcv = 1
	
	Determine Loop Condition
	WHILE lcv <= 100 DO
	
	Update Loop Var
	lcv = lcv + 1
	ENDWHILE
		Label 1:_____
		sum = 0
		lcv = 1
		Label 2:_____
		WHILE lcv <= 100 DO
		
		Label 3:_____
		lcv = lcv + 1
		ENDWHILE

Figure 21: Partial worked example formatted with no labels, given labels, or placeholders for generated labels

Participant groups also received different practice problems to test how contextual transfer may affect learning. In the isomorphic transfer condition, the procedure and context used to solve the worked example and practice problem were exactly the same but the exact values in the problem changed. For example, if a worked example asked participants to find the average of quiz scores with values 70, 80, and 90, then the practice problem asked participants to find the average of quiz scores with values 75, 85, and 95. In the contextual transfer condition, the procedure used to solve the worked example and practice problem were the same except the context of the problem changed. For example, if a worked example asked participants to find the average of quiz scores, then the practice problem asked participants to find the average of money amounts. The contextual transfer was intended to be harder for participants to map concepts from the worked example to the practice problem. More difficult mapping can improve learning by reducing illusions of understanding caused by shallow processing thus inducing deeper processing of information [17, 42, 99]. However it can also increase cognitive load and potentially hinder learning [130].

After completing the instructions, participants completed novel programming tasks to measure their problem solving performance. We hypothesized that students who generated subgoal labels would learn better than those who were given the subgoal labels, and both groups would do better than those who had no subgoals at all. We also hypothesized that learners whose practice problems required contextual transfer would perform better than learners whose practice problems were the same context, unless the contextual transfer required too much cognitive load during the learning process.

5.1.3 Design

The experiment was a 3-by-2, between-subjects, factorial design: the format of worked examples (unlabeled, subgoal labels given, or subgoal labels generated) was crossed with the transfer distance between worked examples and practice problems (isomorphic or contextual transfer). The dependent variables were performance on the pre- and post-test, and

problem solving tasks, both writing code and a Parsons problem.

5.1.4 Participants

Participants were 120 students from two technical universities in the Southeast United States (Table 12). Students were offered credit for completing a lab activity or extra credit as compensation for participation. All students from these courses were allowed to participate, regardless of prior experience with programming or using while loops. To account for prior experience, participants were asked about their prior programming experience in high school (either regular or advanced placement courses) and college and whether they had experience using while loops. Other demographic information collected included gender, age, academic major, high school grade point average (GPA), college GPA, number of years in college, reported comfort with use of a computer, expected difficulty of the programming task, and primary spoken language. There were no statistical differences between the groups for demographic data, which is expected because participants were randomly assigned to treatment groups. Participants also took a multiple-choice pre-test to measure problem solving performance for using while loops. Average scores on the pre-test were low, 1.6 out of 5 points, with 23% (28 out of 120) of participants earning no points.

Many participants did not complete all tasks of the experiment. Participants received compensation regardless of the amount of time or effort that they devoted to the experiment, which might have caused low motivation in some participants. Participants who did not attempt all tasks were excluded from analysis. Participants who answered more than two questions correctly out of the five on the pre-test were excluded from analysis because the instructions were designed for novices.

Table 12: Participant Demographics

Age	Gender	GPA	Major
M = 21.6	71% male	M = 3.2 / 4	52% CS major

5.1.5 Procedure

An outline of the entire study is given in Figure 22. After granting consent (Step 1), the participants completed a demographic questionnaire (Step 2) and pre-test (Step 3). The pre-test was comprised of multiple choice questions about indefinite (while) loops from previous Advanced Placement Computer Science exams. Because the questions were multiple-choice, participants needed to only recognize correct answers rather than create correct answers. When participants finished the demographic questionnaire and pre-test, they began the instructional period (Steps 4-6). The instructional period started with training. Participants who generated their own subgoal labels received training on how to create subgoal labels. The training included expository instructions about generating subgoal labels and an example of a subgoal labeled worked example similar to that in Figure 21. Then the training asked participants to complete activities to practice generating subgoal labels.

The first activity asked participants to apply the subgoal labels from the example to a new worked example. The second activity asked participants to generate their own subgoal labels for an order of operations math problem. After participants generated their own subgoal labels, they were given labels created by an instructional designer for comparison.

Participants who did not generate their own subgoal labels received training to complete verbal analogies. Verbal analogies (e.g., water : thirst :: food : hunger) were considered a comparable task to subgoal label training because they both require analyzing text to determine an underlying structure. Participants who were not asked to generate their own labels were not given subgoal label training because it might have prompted them to process the instructions more similarly than would be expected to participants who were asked to generate their own labels, which might confound the results. Like the subgoal label training, the analogy training included expository instructions, worked examples, and activities to carry out.

Following the training, the instructional period provided worked examples and practice problem pairs (Step 6) to help participants learn to use while loops to solve problems. The

Step	No Subgoal Labels		Subgoal Labels Given		Subgoal Labels Generated	
1	Consent					
2	Demographics					
3	Pre test					
4	Training Problem – summing					
5	Analogy Training & Activity				Subgoal Training & Activity	
Groups	None-Isomorphic	None-Context Transfer	Given-Isomorphic	Given-Context Transfer	Generate-Isomorphic	Generate-Context Transfer
6	Worked Example 1 (no subgoal labels)		Worked Example 1 (subgoal labels given)		Worked Example 1 (space to generate subgoal labels)	
	Problem 1 (no subgoal labels)	Problem 1A (no subgoal labels)	Problem 1 (subgoal labels given)	Problem 1A (subgoal labels given)	Problem 1 (space to generate subgoal labels)	Problem 1A (space to generate subgoal labels)
	Worked Example 2 (no subgoal labels)		Worked Example 2 (subgoal labels given)		Worked Example 2 (space to generate subgoal labels)	
	Problem 2 (no subgoal labels)	Problem 2A (no subgoal labels)	Problem 2 (subgoal labels given)	Problem 2A (subgoal labels given)	Problem 2 (space to generate subgoal labels)	Problem 2A (space to generate subgoal labels)
	Worked Example 3 (no subgoal labels)		Worked Example 3 (subgoal labels given)		Worked Example 3 (space to generate subgoal labels)	
	Problem 3 (no subgoal labels)	Problem 3A (no subgoal labels)	Problem 3 (subgoal labels given)	Problem 3A (subgoal labels given)	Problem 3 (space to generate subgoal labels)	Problem 3A (space to generate subgoal labels)
7	Cognitive Load Measurement					
8	Problem Solving Assessment (4 problems; 2 near transfer, 2 far transfer)					
9	Assessment Task 2					
10	Assessment Task 3					
11	Post Test					

Figure 22: Study Outline

worked example format differed between subjects among three levels: unlabeled, subgoal labels given, and subgoal labels generated. Furthermore, the transfer distance between worked example and practice problem differed between subjects between two levels: isomorphic or contextual transfer. For a summary of the procedure during the instructional period, please refer to Table 22.

Having completed the instructional period, participants were then asked to complete

a 10 item survey designed to measure cognitive load as described in the previous chapter ([91]). The placement of the cognitive load survey at this point is to ensure measurement of the actual learning process and not the assessment elements.

Once participants completed the cognitive load survey, they started the assessment period (Steps 8-11). The assessment period included three types of tasks: 1) a problem solving task which asked the participant to write code (step 8); 2) an Explain in Plain English task where the participant was presented with the correct solutions to the practice problems they were given and asked to group steps together and label the groups (step 9); and 3) a Parsons problem assessment (step 10).

The problem solving / code writing task asked participants to use the problem-solving structure that they had learned during the worked example-practice problem pairs to solve four novel problems. Two of these problems required contextual transfer, meaning that they followed the same steps found in the instructions but in a different context, or cover story. The other two problems required both contextual and structural transfer. In these problems the context was new to the participants and the solution to the problem required a different structure than the problems found in the instructional material (e.g., the practice problem is summing values, the assessment is counting matching values). These tasks were intended to measure participants' problem solving performance as a 'far' transfer.

The second assessment task was to explain in plain English a given code segment. Participants were given the correct solution to the four assessment problems and asked to group the statements into cohesive groups and then give a name or 'explanation' for that group of statements. Analysis of this assessment task is beyond the scope of this research and will not be discussed.

The Parsons problem used for assessment was a version of the "rainfall problem" [59]. The problem had 13 different code pieces with between 1 and 3 steps in each code piece. The participants were asked to put the code steps in order with no consideration of indentation. In other words, they indicated the order of the code segments by numbering them.

After the assessment period, participants completed a post-test that had the same questions as the pre-test to measure their learning (Step 11).

We collected process data throughout the instructional period. We collected performance on the training activities and practice problems to ensure that participants were completing tasks. We also collected the labels that participants created.

We entered into the study with the following hypotheses:

H1. Participants who learn with subgoal labels (given or generated) will perform better on programming assessments and a post-test.

H1A. Those who generate their own subgoal labels and receive multiple variations of the problems (contextual transfer condition) will perform the best on the assessments, unless dealing with transfer overloads their mental resources.

H1B. Participants who learn with subgoal labels (given or generated) will perform better on low cognitive load assessments.

H2. Participants who generate subgoal labels will perform better on problem solving tasks that require farther transfer. Those groups exposed to contextual transfer practice problems will perform better on transfer tasks than the isomorphic transfer groups.

H3. Changing the context or "cover story" between the worked example and practice problem should have limited effect on student performance on Parsons problem assessments.

Notice that H1 from this study maps directly to research question 2 of the dissertation, **RQ2: Does introducing subgoals (either given or learner generated) result in better learning performance?**. Mappings of the individual hypotheses can be found in Table 13.

5.2 Data Analysis

5.2.1 Accuracy of Programming Assessment

We scored participants' solutions for accuracy to generate a problem solving score. Participants earned one point for each correct line of code that they wrote. This scoring scheme

Table 13: Mapping of Study Hypotheses to Research Questions

Hypotheses From Study	**Research Questions from Dissertation**
H1. Participants who learn with subgoal labels (given or generated) will perform better on programming assessments and a post-test.	H2A: Learning activities with subgoals result in better learning performance than those without subgoals.
H1A. Those who generate their own subgoal labels and receive multiple variations of the problems (contextual transfer condition) will perform the best on the assessments, unless dealing with transfer overloads their mental resources.	H2B: Students who generate subgoals exhibit better learning performance than those who are given subgoals.
H1B. Participants who learn with subgoal labels (given or generated) will perform better on low cognitive load assessments.	
H2. Participants who generate subgoal labels will perform better on problem solving tasks that require farther transfer. Those groups exposed to contextual transfer practice problems will perform better on transfer tasks than the isomorphic transfer groups.	
H3. Changing the context or "cover story" between the worked example and practice problem should have limited effect on student performance on Parsons problem assessments.	H2C: A lower cognitive load assessment activity can provide evidence of learning that a high cognitive load assessment does not.

allowed for more sensitivity than scoring solutions as wholly right or wrong. If participants wrote lines that were conceptually correct but contained typos or syntax errors (e.g., missing a parenthesis), they received points. We scored logic errors (having < rather an <=) as incorrect. We considered scoring for conceptual and logical accuracy more valuable than scoring for absolute syntactical accuracy because participants were still early in the learning process. Participants could earn a maximum score of 44.

The effect of the interventions on problem solving performance depended on the worked example manipulation (see Figure 23). We found a statistically significant main effect of worked example format, F (2, 114) = 5.07, MSE = 176.5, p = .008, est. ω^2 = .08, f = .21. To explore this result, we conducted a post-hoc analysis with the LSD test because

it is the most powerful for comparing three groups. We found that both subgoal-oriented formats (i.e., given or generate subgoal labels) performed better than the unlabeled group, mean difference = 7.8, p = .01, and mean difference = 8.6, p = .005, respectively. Both subgoal-oriented formats performed equally, mean difference = .78, p = .80. For transfer distance, we found no main effect, F (2, 114) = 0.42, MSE = 176.5, p = .52, est. ω^2 = .004. These findings are tempered by an interaction between the two interventions.

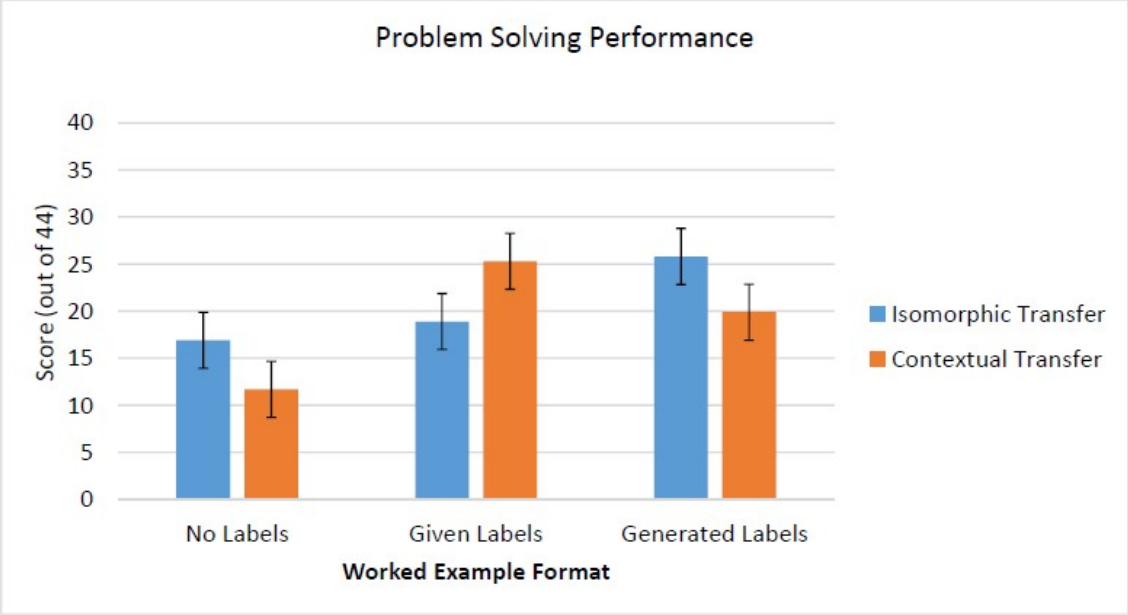

Figure 23: Problem solving performance graphed with worked example format on the x-axis, transfer distance as separate colors, and score on the y-axis

We found a small, but interesting, interaction between the format of worked examples and practice problems and the transfer distance between them, F (2, 114) = 2.71, MSE = 176.5, p = .071, est. ω^2 = .05, f = .15. Though this interaction does not pass the threshold for statically significant in the null hypothesis significance testing framework, the size of the effect makes it worth discussion. We found three levels of performance, as can be seen in Figure 23. The best performing groups were those that were given subgoal labels with contextual transfer (M = 25.3) and generated subgoal labels with isomorphic transfer (M = 25.8). The middle groups were those that received no subgoal labels with isomorphic

transfer (M = 16.9), received labels with isomorphic transfer (M = 18.9), or generated subgoal labels with contextual transfer (M = 19.9). The worst performing group received no subgoal labels with contextual transfer (M = 11.7).

Each level of performance is separated by about seven points, or 16% of the total score. The difference between the middle and best level of performance was not statistically significant but had a medium effect size, as shown by the t-test comparing groups that were given subgoal labels, t(38) = 1.45, p = .15, d = .46. Similarly, the difference between the middle and worst level of performance was not statistically significant but had a medium effect size, as shown by the t-test comparing groups that received labels with isomorphic transfer and that did not receive labels with contextual transfer, t(38) = 1.73, p = .09, d = .65. Given these effect sizes, we would expect these differences to be statistically different with a sample size that was larger than 20 participants per group.

5.2.2 Accuracy on Parsons Problem

We scored participants' Parsons problem answers for correct order to create their score. Participants ranked the 13 code pieces from the Parsons problem and we gave them one point for each code piece that was in the correct order relative to the pieces around it. For example, if participants ranked the 4th, 5th, and 6th pieces of the problem as the 5th, 6th, and 7th pieces of their solution, they would receive two out of three possible points for those three pieces. The first piece would be counted as wrong because it is not following the 3rd piece, but the other two pieces would be counted as correct because they are following the correct piece. This scoring scheme better captures participants' understanding than scoring for absolute correct order as it does not penalize correct sequences of code that follow incorrect sequences.

The effect of the interventions on Parsons problem performance depended on the worked example manipulation (Figure 24). Participants who were given subgoal labels in the worked example performed better than those who generated their own labels or were

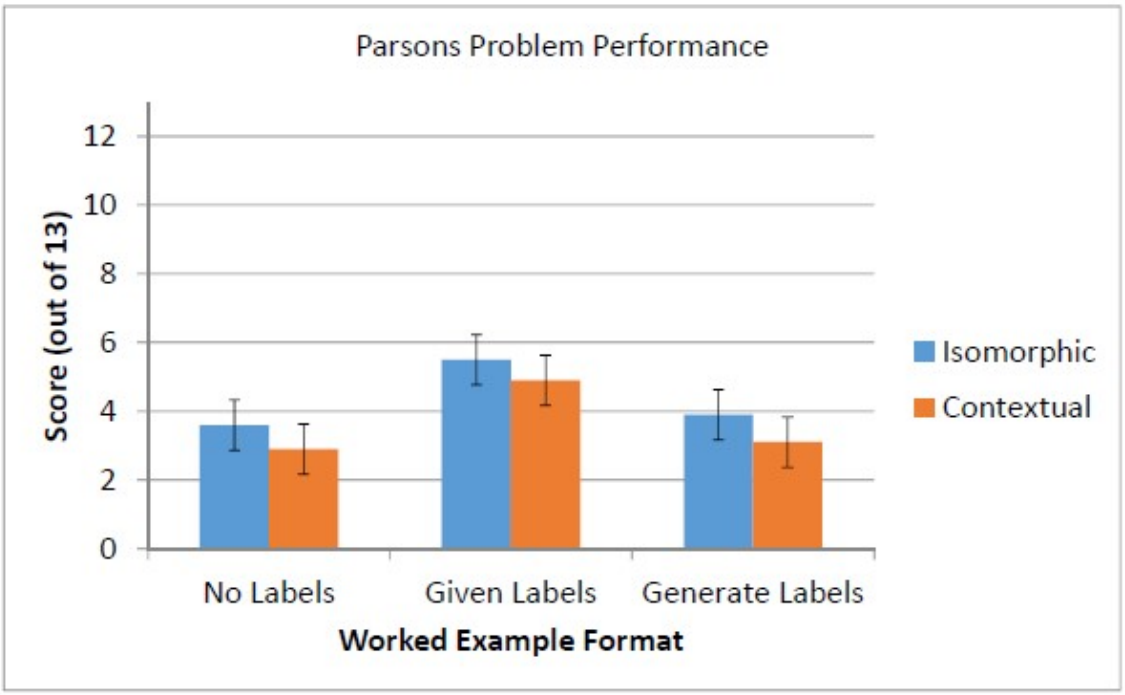

Figure 24: Parsons Problem performance graphed with worked example format on the x-axis, transfer distance as separate colors, and score on the y-axis

not given labels, F (2, 113) = 4.0, MSE = 42.4, p = .021, est. ω^2 = .07, f = .18. We found no main effect of transfer distance, F (2, 113) = 1.3, MSE = 13.9, p = .256, est. ω^2 = .012. We also found no interaction between worked example format and transfer distance, F (2, 113) = 0.006, MSE = .06, p = .994, est. ω^2 = .000.

5.2.3 Post Test and Demographics

Performance on the post-test was similar to that on the pre-test. Average scores on the post-test were low, 34% (1.7 out of 5 points). We found no statistical differences for main effect of worked example format, F (2, 101) = 1.40, MSE = 2.1, p = .25, est. ω^2 = .03, main effect of transfer distance, F (2, 101) = .26, MSE = .39, p = .61, est. ω^2 = .003, or interaction, F (2, 101) = .105, MSE = .158, p = .9, est. ω^2 = .002.

Some demographic characteristics correlated with performance on the problem solving tasks. Having taken a high school computing class (r = .224, p = .014) or an AP

CS high school class (r = .267, p = .003) both correlated positively with problem solving performance. Self-reported comfort with solving programming problems, collected on a Likert-type scale from "1 –Not at all comfortable" to "7 –Very comfortable," correlated positively with performance, r = .47, $p < .001$. Prior experience using while loops to solve programming problems, collected as a "yes" or "no" question, correlated positively with performance, r = .29, p = .018. Higher scores on these characteristics correlated with higher scores on performance. We found no differences among groups on these characteristics; thus, these correlations are not expected to confound the results.

5.3 Results

5.3.0.1 Assessments

Three groups performed the best on the assessments —combining the programming assessment and post test: those that were given subgoal labels with contextual transfer (*Given-Context Transfer*), and both groups that generated subgoal labels (*Generate-Isomorphic* and *Generate-Context Transfer*) (Figure 25).

Interestingly, the *Generate-Context Transfer* group did better on the post-test while the *Generate-Isomorphic* group performed better on the programming assessments. However the group that was given subgoal labels with no contextual transfer performed relatively poorly on both the programming assessment and the post-test.

Thus we have partial support for H1. We found that both subgoal-oriented formats (i.e., given or generate subgoal labels) performed better than the unlabeled group at a statistically significant level for the programming assessment task. For the related hypothesis H1A, it was the case that the *Generate-Context Transfer* group performed non-significantly better on the programming assessment task; they did not outperform the other groups on the Parsons problem or post test. So H1A has partial support. This may be because the generation of subgoal labels while also considering the contextual transfer overloaded the participants during the learning acquisition phase or the assessment phase.

When the commonalities between worked examples and practice problems were evident, as in the isomorphic transfer conditions, generating subgoal labels might have encouraged deep processing of information without overloading the participants. Similarly, when subgoal labels are given to participants, finding commonalities between contextually different examples and problems might have encouraged deep processing of information without overloading the participants. Participants who both generated subgoal labels and had contextual transfer did not perform as well as these groups. It is possible that both generating subgoal labels and finding commonalities between contextually different worked examples and practice problems was too cognitively demanding for many of the participants, which hindered performance.

We found that students who were given subgoals performed statistically significantly better than those who had no subgoals or who generated their own subgoals, regardless of transfer condition, on the Parsons problem. In other words, both the *Given-Isomorphic* and *Given-Contextual Transfer* groups performed statistically better than the other groups. In all cases the isomorphic groups did better than their contextual transfer counterpart, however these differences were not significant. We thus have partial support for H1B: Participants who learn with subgoal labels (*given only*) will perform better on low cognitive load assessments. Because there were no statistical differences between the isomorphic problem groups and the contextual transfer groups, we have support for H3, transfer appears to have limited effect on student performance on this task.

5.3.0.2 Transfer Tasks

The best performing group on the transfer tasks (programming assessments 3 and 4) was the group that was given subgoal labels and contextually different practice problems (*Given-Context Transfer*) (see Figure 26). However the other two groups receiving contextual transfer practice problems did not perform particularly well on the transfer programming tasks and nothing was statistically different.

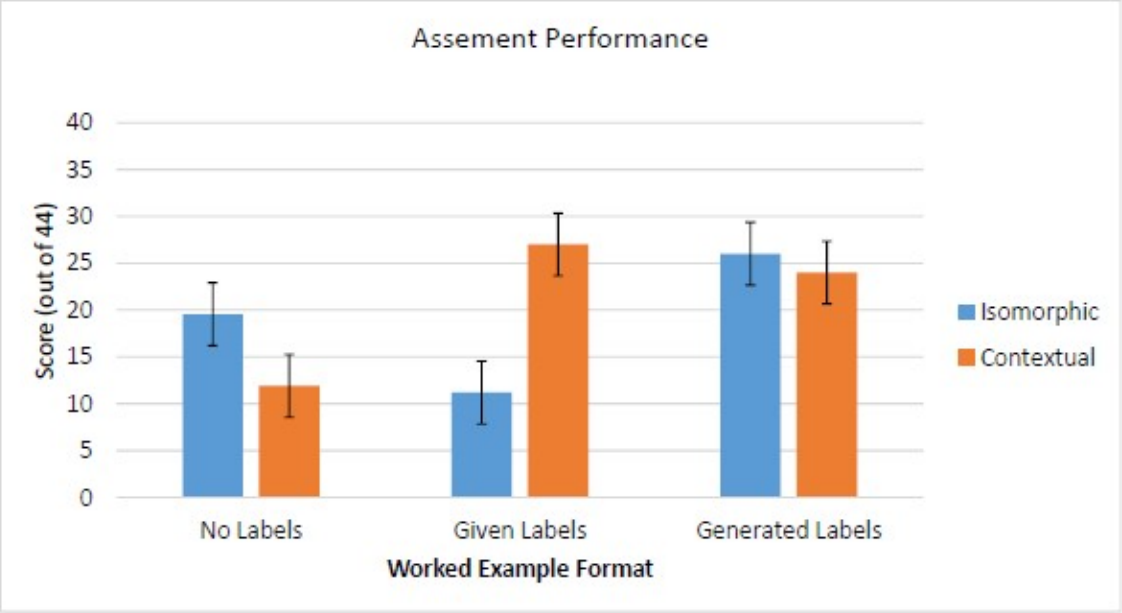

Figure 25: Total assessment performance graphed with worked example format on the x-axis, transfer distance as separate colors, and score on the y-axis

So we find no support for H2. Those groups who were exposed to contextual transfer problems did not perform better than their isomorphic problem counterparts and this included the group that generated their own subgoal labels. However it should be noted that it was a contextual transfer group that did perform the best on the far transfer tasks, those that were given the subgoal labels.

5.4 Contributions

Groups that generated subgoal labels performed overall better than those that did not have subgoal labels. The pattern of results for these groups is similar, though. In both cases, the condition that had isomorphic problems performed better than the condition that had contextual transfer, quite possibly because solving the isomorphic problems required less cognitive load. This pattern is reversed for groups that were given subgoal labels. It might be the case that learners who contend with contextual transfer problems need help identifying the analogous subgoals of the worked examples and practice problems. Participants

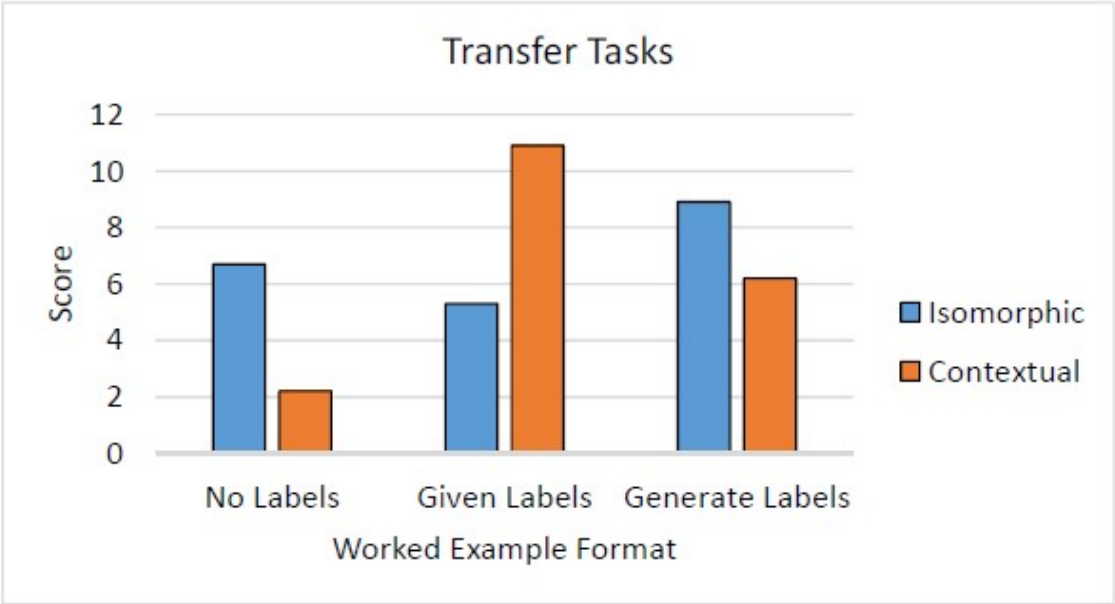

Figure 26: Tranfer problem solving assessment performance graphed with worked example format on the x-axis, transfer distance as separate colors, and score on the y-axis

who were given subgoal labels with contextual transfer might have been one of the highest performing groups because they received a framework of meaningful subgoal labels that guided their transfer between worked examples and practice problems. Though participants who generated subgoals labels received placeholders that indicated analogous subgoals between examples and problems, some of their generated labels were context-specific to the problem, which would not likely promote transfer to a contextually-different problem. In addition, if participants were unsure of the labels that they generated, they might rely less on them to guide future problem solving. Students that generated abstract labels performed the best on the assessments regardless of their prior knowledge, differing from the results found in [25].

The most surprising result from this experiment was the group that was given subgoal labels and isomorphic problems was one of the worst performing groups. It could be that being given the labels in addition to being able to more easily recognize commonalities

between worked examples and practice problems led to superficial processing of information. Because participants could solve practice problems by using the worked example as an isomorphic guide and because the subgoal labels explained the function of programming steps, participants might have been overconfident about their understanding of the procedure and devoted less effort to learning.

Participants that were given subgoal labels performed statistically better than those that did not have subgoal labels and those that generated their own subgoal labels on the low cognitive load assessment task, the Parsons problem. It appears that for generating code solutions, generating subgoal labels encourages the most learning. However just being given subgoal labels allows learners to grasp the solution process which can be tested using the low cognitive load assessment technique of Parsons problems. Low cognitive load learning did not always lead to better learning performance on high cognitive load assessments like writing code. We can take the following implication from these findings. Subgoal labels can reduce cognitive load which allows student to focus and learn more efficiently. Students who are given subgoal labels while learning problem solving can most likely recall those labels when needed to arrange code segments into order, the order of the learned subgoal labels. Previous work has shown that learning subgoal labels also helps with transfer [75]. The subgoal labels provide structure for organization of student learning.

Changing the context between the worked example and the practice problem had limited effect on the results of performance on the Parsons problem. This may be explained by cognitive load. We know that adding transfer between the worked example and practice problem introduces additional cognitive load–students must do non-superficial transfer during learning. This additional cognitive load could alter the learning just enough so that it shows up in a high cognitive load task (writing code from scratch) but not in a low cognitive load task (Parsons problems). Students who can learn the appropriate order of the subgoals may be able to demonstrate that knowledge on Parsons problems, where they may not be able to do so when writing code from scratch.

We collected and analyzed cognitive load component measurements using [91], however the differences were not statistically significant. No group reported significantly higher cognitive load, even though we know that generating subgoal labels requires more thought and mental effort than just reading and understanding given subgoal labels. Likewise, contextual transfer had no effect on the cognitive load component measures. This may be explained because all conditions had the same amount of intrinsic load, or because the measurement tool is not sensitive enough to capture the differences in this instance. This is definitely an area that needs further exploration.

5.4.0.1 Summary of Findings

See Table 14 for a complete summary of our findings.

Notice that the hypotheses that were related to research question 2 were answered. We have limited support that introducing subgoals (either given or learner generated) does result in better learning performance. Specifically, learning activities with subgoals result in better learning performance than those without subgoals. Statistically, we found that both subgoal-oriented formats (i.e., given or generate subgoal labels) performed better than the unlabeled group, with context transfer having no effect. We also found that participants who were given subgoal labels in the worked example performed statistically better than those who generated their own labels or were not given labels on the Parsons problem assessment task.

5.5 Limitations

The interventions for this study are strongly grounded in instructional design theory, and they were also applied in an authentic educational setting with an authentic educational task. Therefore, we expect that the internal and external validity of this work is high. However, because this study is the first experiment to use this type of task and because the results were different than previous work with subgoal labels, research to replicate these results is needed to ensure the validity of this work.

Table 14: Findings from study

Hypotheses From Study	**Findings**
H1. Participants who learn with subgoal labels (given or generated) will perform better on programming assessments and a post-test.	Partially supported Groups who learned with subgoal labels performed better on problem solving assessment but no statistical difference on post-test
H1A. Those who generate their own subgoal labels and receive multiple variations of the problems (contextual transfer condition) will perform the best on the assessments, unless dealing with transfer overloads their mental resources.	Generate groups performed better on problem solving assessment but not on the post test or Parson problem assessment.
H1B. Participants who learn with subgoal labels (given or generated) will perform better on low cognitive load assessments.	Supported. Given labels groups performed the best on low cognitive load assessment.
H2. Participants who generate subgoal labels will perform better on problem solving tasks that require farther transfer. Those groups exposed to contextual transfer practice problems will perform better on transfer tasks than the isomorphic transfer groups.	Not supported.
H3. Changing the context or "cover story" between the worked example and practice problem should have limited effect on student performance on Parsons problem assessments.	Supported. Contextual transfer had no effect on performance on Parsons problem assessment.

5.6 Discussion

Our findings continue to support the belief that subgoal labeling does improve learning. Generating those labels takes more time, and more time does result in more learning. However, being given labels may result in about the same amount of learning. In terms of efficiency (the most learning for the least amount of resources, including time), being given the subgoal labels may be the best option.

Having a context shift, from the example to the practice problem, appears to help those who are given subgoal labels. This may be the best case for allowing the learners to generalize across examples without imposing too much cognitive load. The contextual transfer

for the group that was required to generate subgoal labels appears to have had their learning performance decrease, perhaps due to the extra cognitive load resulting from the context shift. The best performance on the assessments comes from giving students the subgoal labels and requiring contextual transfer, or having students generate the subgoal labels but using only isomorphic transfer from example to practice.

The problem is that cognitive load in computer science is high due to the intrinsic nature of the material. Students have to keep in mind variables, their roles, their own process in problem-solving, and the process of the computer that they are attempting to model and control. While generating subgoal labels intuitively should lead to greater learning, there comes a point (e.g., if we add in contextual transfer) when the cognitive load of tracking everything makes learning difficult.

The intrinsic cognitive load of computer science is related to the languages we use (e.g., the fact that textual languages require naming of data and process, and we must remember and use those names) and the challenge of understanding and controlling a computational agent other than ourselves. That kind of problem does not occur frequently in science, mathematics, and engineering —but occurs from the very first classes in computer science. Because of this intrinsic load and the differences from other disciplines, we need to conduct replication studies. We cannot simply assume that findings from these other disciplines will predict learning in computer science.

5.7 Replication[2]

To explore whether the original study was an anomaly, a replication study with a different population at a different institution was conducted. We found that the original results were not an anomaly, they repeated with this new population. In the original study, the average score on the post-test, which comprised items from the AP CS test, was only 31%, indicating that the students had not learned very much from the intervention. Perhaps the

[2]This work was done with the assistance of Adrienne Decker and Lauren Margulieux who helped with collection of data and statistical analysis results.

assessment tasks were too difficult and once again we were asking more from our students than they were capable of (as seen in [85]). To explore whether the assessment tasks were too difficult, students in a follow-on programming course were asked to participate in the study. We compared student performance in the follow-on programming course with those in the introductory course. We found that students in the follow-on course performed better on the assessments than the introductory students, indicating that writing while loops can be mastered. In addition, we found that students in the introductory programming course who had taken computing courses in high school performed better than those who did not. However, this difference in performance due to high school experience was not present for students in the follow-on course.

5.7.0.1 Method

Participants in introductory programming classes who had already been introduced to loops within their course were given additional instructional material designed to reinforce the practice of solving programming problems using while loops. Participants were recruited from 3 different first and second year programming courses at a technical university in the northeast United States and the study was conducted over a one month period.

Table 15 summarizes the differences between the three courses. The first two courses are first year, first semester courses serving primarily two different populations of students. The first course (101) serves as the first programming course (CS1 equivalent) for students intending to major in New Media Interactive Design (a College of Imaging Arts and Sciences major), or New Media Interactive Development (a College of Computing and Information Sciences major) and is taught using Processing. The "New Media" majors are focused on the interaction of art and technology through media. The difference between the students is the focus of the major, the "design" major attracts primarily students who may consider themselves artists, while the "development" major attracts those who are more "technologists". The second course (105) serves as the first programming course (CS1

equivalent) for students intending to major in Game Design and Development (a College of Computing and Information Sciences major) and is taught in C#. The game design and development degree is a technically focused degree in game design and development and the coursework has many similarities to a computer science degree. The department does not give credit for either 101 or 105 for Advanced Placement (AP) credit. Students earning high scores on the AP exam earn credit for another course from another department, but still need to take 101 or 105 to complete the requirements for their respective majors.

Table 15: Classes Participating in Study

Course	**Programming Language**	**Majors**	**Experiment Delivery Method**
101	Processing	New Media Interactive Design, New Media Interactive Development	Closed lab in-class exercise
105	C#	Game Design & Development	Optional at-home assignment
202	C# (some Processing)	New Media Interactive Design, Game Design & Development	Closed lab in-class exercise

The second year course (202) is designed to bring together the groups from New Media Interactive Development and Game Design and Development and is taught primarily in C# (with some limited time devoted to Processing) and focuses on the use and integration of media and media artifacts into interactive experiences. It should be noted that students who take the 101 course take 2 more courses (102 and 201) before taking the follow-on course while those who take the 105 course take only 1 more course (106) before taking 202. So students from the 101 course path have a 3 semester sequence while those in the 105 track have a 2 semester sequence. The study was conducted either in a closed lab setting with up to 30 computers in a single room, or as an optional at-home assignment

(see Table 15). The participating instructors decided how to structure the exercise in their particular course and what weighting it had on a student's grade, but participation in the study was strictly voluntary. That is, even in classes where there was a closed lab around the exercise, participation in the study described here was voluntary; consent to use the data for the study was given at the end of the exercises. All of these courses are taught in a computer lab of at most 30 students. Exercises where students are given a set of tasks to perform during the class period for their grade are a common part of these courses.

Students received an introduction to the study explaining that the material in the study was designed to help them learn how to write loops. Students were then given a URL to the first page of the study, which was housed in SurveyMonkey. Participants worked independently. The in-class sessions were an entire class period for the course (110 minutes). For the students who completed at home, the assignment was posted for them and they were given a due date by which they needed to complete the exercise. At the end of the window, the SurveyMonkey materials were closed.

The materials used were identical to those used in first study reported in this chapter, other than placement of the consent.

5.7.0.2 Participants

Participants were 100 students from a technical university in the northeast United States (Table 16). To account for prior experience, participants were asked about their prior programming experience in high school (either regular or advanced placement courses) and college and whether they had experience using while loops. Other demographic information collected included gender, age, academic major, high school grade point average (GPA), college GPA, number of years in college, reported comfort with computer, expected difficulty of the programming task, and primary language. There were no statistical differences between the groups for demographic data, which is expected because participants were randomly assigned to treatment groups.

Table 16: Participant Demographics

Age	Gender	GPA	Major
M=19	72% male	M=3.5/4	33% New Media 63% Game Design 3% CS, SWE, CEngr

Participants who did not attempt all tasks were excluded from analysis. For the replication piece of the study, participants who answered more than two questions correctly out of the five on the pre-test were excluded from analysis because the instructions were designed for novices. However, for the second piece of analysis within the paper, we looked at the success rate of all students who completed all the tasks. Based on these exclusion criteria, we analyzed data from 27 participants for the replication study and 100 participants for overall performance.

We entered into the replication study with the following research questions: **R1.** Do participants who learn with given subgoal labels and no contextual transfer perform better or worse on programming assessments than those who learn with given subgoal labels and contextual transfer? **R2.** Do participants further along in their computing studies outperform novices on both the pre/post-test and programming assessments?

5.7.0.3 Analysis and Results

For this replication study, we report on findings that support the original study and one additional piece of information gleaned from the study –that students do master writing loops at a later point in their academic careers. In the statistics reported below, we include two types of effect sizes. The first, est. ω^2, describes how much of the variation in scores can be attributed to the manipulation. For example, for the code writing assessment, an est. ω^2 of .10 means that 10% of the variation in performance can be attributed to the instructional manipulations. The second, f or d, describes the difference between groups using the standard deviation as the unit of measurement. For example, for the code writing assessment, a d of .5 would mean that the difference between the means of two groups is

half of the standard deviation for those groups.

The effect of the interventions on code writing performance depended on the interaction of the worked example manipulation and transfer distance manipulation. We found no main effect of worked example format, F (2, 21) = 0.26, MSE = 105.6, p = .78, est. ω^2 = .02. In addition, we found no main effect of transfer distance, F (1, 21) = 1.47, MSE = 105.6,p = .24, est. ω^2 = .07. There was, however, a statistically significant interaction between worked example format and transfer distance, F (2, 21) = 5.19, MSE = 105.6, p = .015, est. ω^2 = .33, f = .44 (see Figure 27).

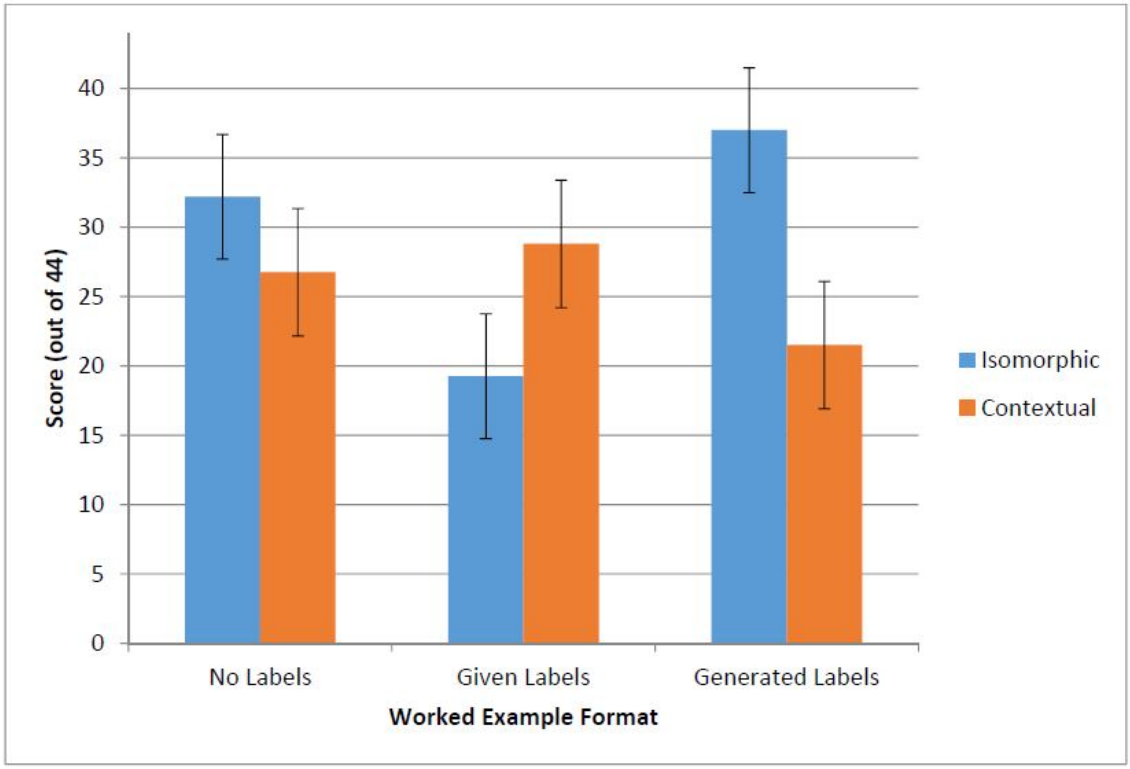

Figure 27: Code writing performance for novice programmers [3]

In this interaction, the difference between the group that was given subgoal labels with isomorphic transfer (M = 18.4, SD = 13.0) and the group that was given subgoal labels with contextual transfer (M = 31.2, SD = 13.8) was not statistically significant, t (8) = -1.50, p

[3]Error bars on all bar graphs represent the 95% confidence interval.

= .17, but the difference between groups was large with an effect size of d = 0.95. Based on these results, the difference between groups is meaningful, even though it is not statistically significant, likely due to a small sample size. The sample size was small because this was a replication study thus we did not need as much statistical power to ensure the pattern of results was reliable. Because the effect size is large and matches previous results we conclude that this is a meaningful difference. Furthermore, the difference between the group that generated subgoal labels with isomorphic problems (M = 18.0, SD = 4.6) and the group that generated subgoal labels with contextual transfer (M = 37, SD = 4.5) was statistically significant, t (4) = 7.18, p = .002, with a large effect size, d = 4.18. This effect size is based on a sample of six, therefore, it is likely not reliable. The effect size is likely inflated due to the small sample; however, the effect of the intervention is still valid and in the correct direction. These results mean that participants who were given subgoal labels performed better when they had contextual transfer, and participants who generated subgoal labels performed better with isomorphic problems. These results match the previous study.

As mentioned earlier, we also asked students in a follow-on programming course to participate in the study. This section reviews their performance and compares it with the novice performance. For this analysis we looked at all students who completed the tasks regardless of their pre-test score.

Students considered for the results in Figure 27, the replication study, had to match the qualifications of the original study. This meant that we excluded all introductory students that correctly answered 3 or more questions on the pre-test correctly. This eliminated 24 students, almost as many as we analyzed (27). It became clear that many of the students in the introductory courses had significant loop writing knowledge prior to our intervention. This led us to further investigate if this prior knowledge could be attributed to prior course-work. Looking at all participants in the introductory courses (n = 51), the average score on the pre-test was 46% (2.3 out of 5). Participants scored about the same on the post-test

with an average of 54% (2.7 out of 5).

Within this group, no manipulation by itself made a statistical difference in code writing performance. There was no main effect of worked example format, F (2, 45) = .32, MSE = 106.1, p = .73, est. ω^2 = .01. There was also no main effect of transfer distance, F (1, 45) = 1.88, MSE = 106.1, p = .18, est. ω^2 = .04. There was, however, an interaction, F (2, 45) = 4.04, MSE = 106.1, p = .024, est. ω^2 = .15 (see Figure 28). This interaction resembles the pattern of results seen in both the original subgoal study and the replication piece of this study.

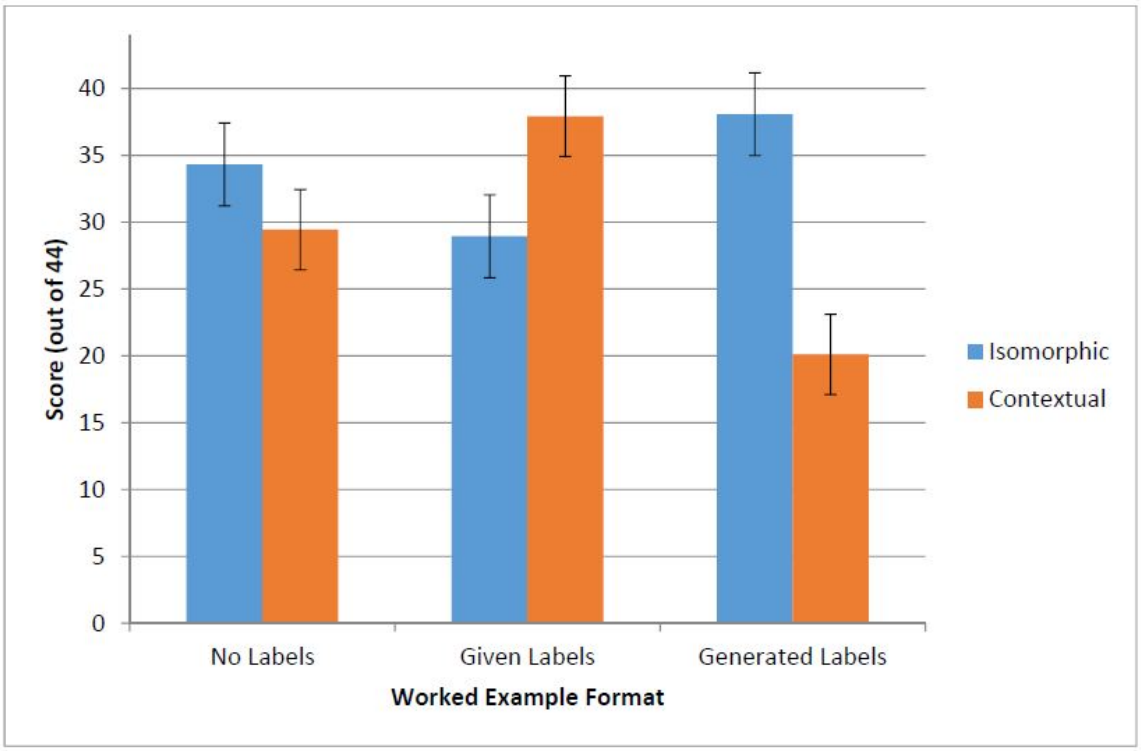

Figure 28: Code writing performance for participants in introductory courses

Participants in the follow-on course (i.e., the course after the introductory college course) were excluded from previous analyses because they had at least a semester of programming instruction, and the instruction in the study was designed for novices. Participants in this course scored on average a 73% (3.6 out of 5) on the pre-test. After instruction

on the post-test, participants scored about the same with an average of 70% (3.5 out of 5). Within this group, participants who were given subgoal labels performed better on the code writing assessments than those who were not given labels or those who generated labels, F (2, 43) = 7.33, MSE = 15.9, p = .002, est. ω^2 = .25, f = .37. There was no main effect of transfer distance, F (1, 43) = .25, MSE = 15.9, p = .62, est. ω^2 = .01, nor was there an interaction, F (2, 43) = 2.30, MSE = 15.9, p = .11, est. ω^2 = .10 (see Figure 29).

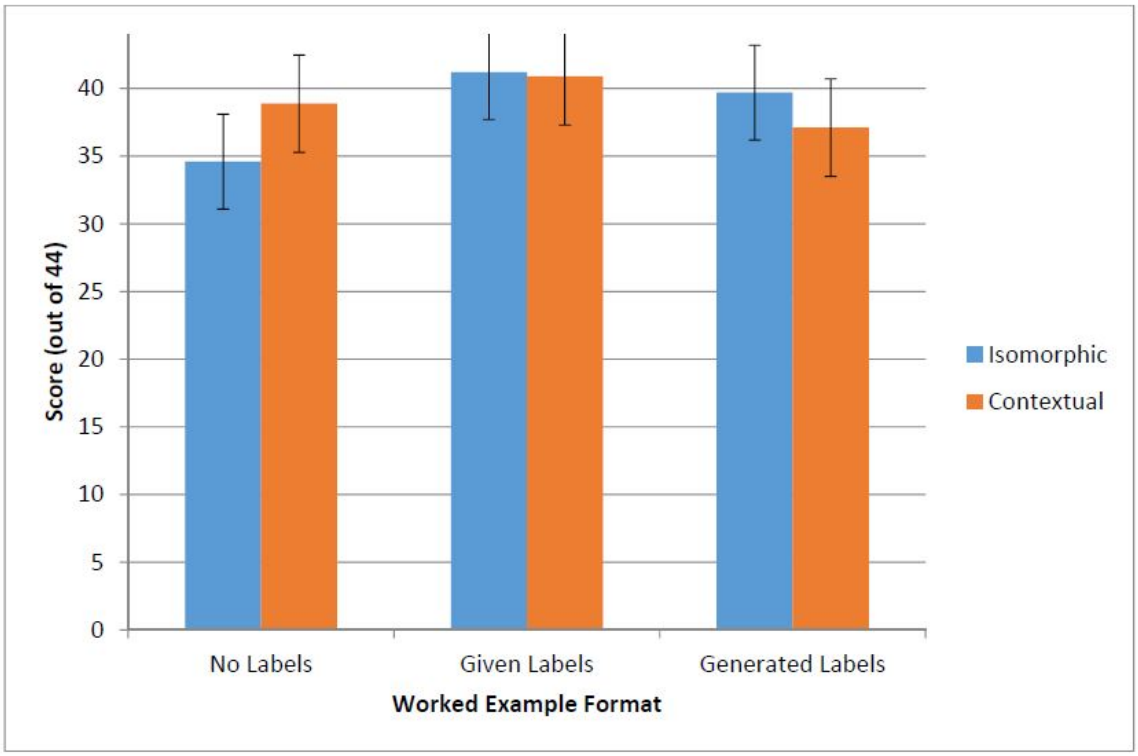

Figure 29: Code writing performance for 202 students

To explore the effect of computing courses in high school and the effect of prior computing courses in college on performance, we used these two variables as random independent variables in ANOVA to determine if they affected performance. Participants who took computing courses in high school performed better on the code writing assessment than those who did not, F (1, 96) = 12.0, MSE = 56.2, p = .001, est. ω^2 = .11, f = .35. Participants who had taken prior computing courses in college performed better, F (2, 96) = 14.3, MSE

= 56.2, $p < .001$, est. $\omega^2 = .13$, f = .38. There was an interaction, F (2, 96) = 11.1, MSE = 56.2, $p = .001$, est. $\omega^2 = .10$, f = .33, such that participants who took computing courses in high school did not perform better that those who did not in later college computing courses (Figure 30). In other words, it does not matter if the previous course was taken in high school or college âĂŞ the fact that the student had a previous course predicts better performance; but that advantage does not continue into the next course.

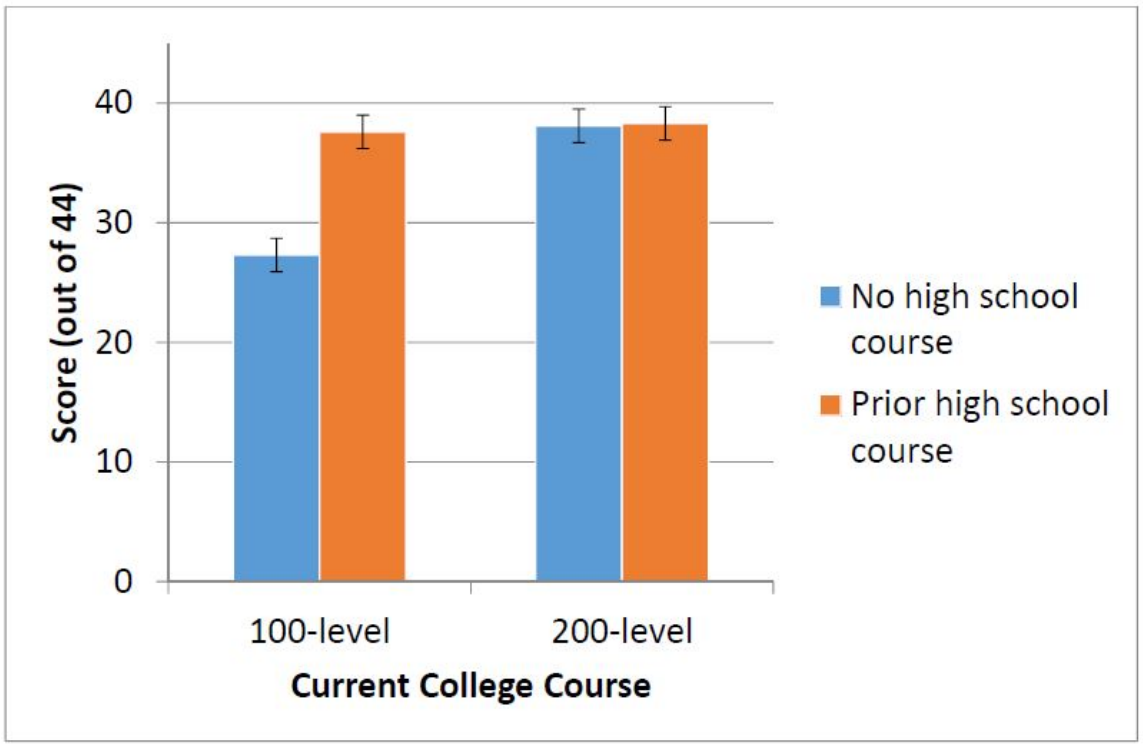

Figure 30: Code writing performance by course level

Similar to the code writing task, participants who took computing courses in high school performed better on the Parsons problem than those who did not, F (1, 96) = 9.85, MSE = 11.6, $p = .002$, est. $\omega^2 = .09$, f = .31. Participants who had taken prior computing courses in college also performed better, F (2, 96) = 6.78, MSE = 11.6, $p = .011$, est. $\omega^2 = .07$, f = .26. For this assessment, however, there was no interaction, F (2, 96) = 1.66, MSE = 11.6, $p = .20$, est. $\omega^2 = .02$, suggesting that those who had computing courses in high school performed better than those who did not, even after their first computing course in

college.

5.7.0.4 Discussion

Here we summarize the findings related to our original research questions and discuss the implications for computing education.

The replication portion of the study yielded results that support the original findings. This study confirms that novice participants who learn by generating subgoal labels (using isomorphic WE-PP pairs) perform the best, and statistically better than if they had been WE-PP pairs with contextual transfer (Figure 27). We conclude that for the best learning results novice students should be taught to generate their own subgoal labels but be given WE-PPs that are very similar.

We hypothesize that teaching novice students to generate their own subgoal labels does require additional time, both for instruction and for the student during the WE-PP instruction time. Additionally it should be noted that within this experiment participants did not receive any feedback on the appropriateness of their generated labels. To obtain maximum benefit from generating subgoal labels, students should receive feedback on the correctness of their labels. Alternately, similar learning results may be obtained by using given subgoal labels. However, if pre-defined given subgoal labels are used, the WE-PP pairs should utilize contextual transfer to ensure maximum learning. As mentioned earlier, this is contradictory to what would be predicted by CLT. This is certainly one phenomenon that needs further research. It may be that with given subgoal labels and isomorphic problems students do not adequately self-explain the process associated with each subgoal as the steps are identical within both the worked example and practice problem. Just as in the first study, we reviewed student code submissions to ensure that they were not copied from the worked example and they were not. Also the time spent in the instructional period indicates that participants spent similar amounts of time regardless if they received isomorphic or contextual transfer WE-PP pairs.

It may be that with given subgoal labels students require multiple examples for comparison to be able to determine the generalities of the labels and the process. It would be interesting to determine if more examples or additional practice problems would improve the learning performance of these groups. It may also be possible to present the worked example with no contextual story at all âĂŞ just as a simple problem to be solved. If students are presented with a "vanilla" worked example with given subgoal labels followed by a practice problem embedded within a context, would the performance differ? Looking into these and other possibilities are planned as future research areas.

We were surprised at the number of students in the introductory courses that were excluded due to their pre-test scores. In looking at the demographics, we noticed many of the introductory students had a previous computing course in high school and their pre-test scores reflected this prior experience. In examining their performance along with the students in the follow-on course, we found that having that prior coursework experience made a significant difference in the performance on the assessment tasks.

Participants in the introductory courses with high school coursework experience performed statistically better on both the code writing and Parsons problem assessments than students in the same courses without high school computing coursework. These students performed similarly to those in the follow-on course (202). Students in the 202 course performed statistically better on the Parsons problem assessments than the students in the introductory courses, regardless of high school coursework. We have evidence that students with some coursework experience, whether in high school or a previous college class, have actually mastered this concept given the pre-test score above 70% and the performance on the code writing assessment (Figure 29 for 202 students only).

However, as can be seen in Figure 30, students in the follow-on course that did not have a high school computing course performed approximately the same as those who did have a high school computing course on the code writing assessment task. Thus, those without the previous coursework advantage had "caught up" to those who started with more

knowledge. This has significant implications for those teaching introductory programming. If students with prior programming experience are in the same class as those with previous computing coursework, we cannot expect them to perform the same on assessment tasks, especially after a short exposure. The participants in the introductory courses had been exposed to loops in their current courses for approximately 2 weeks and were preparing for a graded test which would include loops. Even without knowing the scores for the exam, we predict that the students with previous computing courses in high school would significantly outperform those without that experience.

We found one additional interesting result in this study. Students in the follow-on course who generated subgoal labels performed statistically worse than those in the same course who were not given labels or those who received labels. This may be an example of the expertise reversal effect [62]. The expertise reversal effect occurs when the learner is presented with information that causes them to think below their automatized schema. The instructional design material, in this case the generation of subgoal labels, uses working memory that would not have been necessary if the learner were just solving the problem. In other words, the participant could have solved the problem without any instructional material at all because of their prior knowledge. The instructional material interfered with their problem solving process. Further research into when subgoal labels should no longer be used with those learning programming should be explored.

5.7.0.5 Conclusion

This study was originally conceived as a means to replicate an existing study to determine if the puzzling results would be confirmed. The data gathered in this study confirms that students who learn with given subgoal labels perform better with contextual transfer between the WE-PP pair than those who received isomorphic WE-PP pairs. While we still have no evidence as to why this occurs, contrary to cognitive load theory, we now know that the result is repeatable and deserves further research to investigate why this group in

computing differs from those in other disciplines.

Because of the number of participants eliminated from the replication analysis we explored reasons for the difference in performance. Students who had previous computing coursework outperformed those without previous coursework in both code writing and Parsons problem assessment tasks. Students with previous computing coursework should be assessed at a different standard than those with minimal time exposure to the concept. However, in the follow-on course this performance difference disappears indicating that those without previous coursework do "catch up". This indicates that the students with differing experience backgrounds can be merged into a single class.

While some may think the results of this paper, students with previous experience perform better, are obvious, we demonstrate that our students actually do eventually learn and master a concept (writing loops) unlike so much previous research [85, 119, 73]. We also provide evidence that any advantage gained through previous coursework disappears, with regard to this introductory concept, in the follow-on course. We find these facts, that our students actually do learn and that students without previous experience can catch up, very encouraging.

CHAPTER VI

LOOP STRATEGY STUDY

The findings from the previous studies lead us to posit that learning to program is a high cognitive load activity. The amount of cognitive load on the learner is impacted by what we teach, how we teach, and how we assess the knowledge. When teaching programming, all the elements are connected and build on previous knowledge, much like mathematics. When solving a programming problem, students must know the problem solution, the programming language syntax (for text based programming languages), and the notional machine (how the computing device will interpret the commands). All of these components are competing for resources in their working memory at the same time.

First an instrument for measuring cognitive load components within introductory programming was developed and initially validated. We have explored reducing the cognitive load by changing the modality in which students receive the learning material. This had no effect on novices' retention of knowledge or their ability to transfer knowledge. We then attempted to reduce the cognitive load by adding subgoal labels to the instructional material. This had some effect on the learning gains under some conditions. Students who learned using subgoal labels demonstrated higher learning gains than the other conditions on the programming assessment task. We also explored using a low cognitive load assessment task, a Parsons problem, to measure learning gains. This low cognitive load assessment task proved more sensitive than the open ended programming assessment tasks in capturing student learning. Students who were given subgoal labels regardless of context transfer condition performed better than those in the other conditions.

In this final study, the actual content of the instructional material has been altered in an attempt to reduce the cognitive load. We know that learning how to write loops is difficult

for students [70]. In the altered instructional material, we are aiming to find a more natural format for students to write loops. Originally documented in Soloway, Bonar and Ehrlich [121], students using a programming language that allows them to "leave" in the middle of a loop exhibit superior learning over those that used a traditional loop structure. Surprisingly this still holds 30 years later, even with the change to object-oriented programming [15].

6.1 Background

When Dijkstra penned his now famous "GOTO Considered Harmful" it began a revolution in how loops were introduced to students. Dijkstra argued for *structured programming* even in the case of the "loop-and-a-half" problem. Dijkstra identified the "loop-and-a-half" problem when discussing the algorithm for sequential search. In the sequential search algorithm two outcomes are possible: the search is exhausted without finding the target or the target is found. The question is how the algorithm should be structured in the case when the target is found: should the loop be exited at that exact point (with a GOTO or other exit from the middle of the loop) or set a boolean flag used in the loop condition and then exit when the top of the loop is reached again (hence the name loop-and-a-half). (See Figure 31 for an example.) Dijkstra argued for a single input into the loop and a single exit out of the loop. He argued against the use of a GOTO statement under any condition, regardless of the name of the statement (GOTO, break, return, etc.). Dijkstra was a strong proponent of structured programming, as defined by a single entry and exit point from every module.

Exit in the Middle Loop	**Traditional Loop (loop and a half)**

`x = 1;`
`while (x < length(list))`
`  if (list[x] == target)`
`  break;  //GOTO`
`  else`
`  x = x + 1;`
`if (x == length(list))`
`  //target not found`
`else`
`  // target found`
	`x = 1;`
	`found = false;`
	`while (x < length(list) && NOT found)`
	`  if (list[x] == target)`
	`  found = true;`
	`  else`
	`  x = x + 1;`
	`if (found)`
	`  //target found`
	`else`
	`  // target not found`

Figure 31: Loop and a Half

Dijkstra's vision was realized in the design of the PASCAL programming language which does not allow any exit from the middle of a loop. Modern programming languages have constructs that are similar to GOTO statements. For example, in C and its derivatives, a break statement exists which exits the current control structure only. There is no need to name or label code lines (as with the GOTO statement) so the code is still easily readable. Many instructors continue with the philosophical restriction introduced in PASCAL, argued for by Dijkstra. They do not teach the break statement within the loop control structure, some even penalize students for using it. We assume they do this out of tradition or in continued support of structured programming.

It is interesting to note why Dijkstra and others were so adamant on using structured programming. The original considerations of having only a single entry and exit point for a loop developed from software engineering concerns. Having only single entry and exit points were considered to improve the maintainability of the code, allowed for easier and more thorough testing, and supposedly improved readability. These characteristics are for those developing software that will be reused or put into production –things valuable for software developers, but not necessarily novice programmers who will never program professionally.

In 1983, Soloway, Bonar, and Ehrlich presented an empirical study showing that students preferred and wrote more correct programs when allowed to exit in the middle of a loop for the well known rainfall problem [121]. Soloway et al. argued that exiting in the middle of the loop was a closer 'cognitive fit' with students' preferred cognitive strategy for solving the problem. When talking about repeating a process, most people refer to idea of "continue until X happens" [100]. Soloway et al. asked students to plan the algorithm for solving the rainfall problem and most students selected a process that would allow them to leave in the middle of the loop. In other words, they wanted to continue to read the input data until they encountered the sentinel value. The testing for the sentinel value occurred in the middle of the loop, and if found, the students wanted to exit from the loop immediately,

they did not want to loop back to the top and attempt to read another value.

Soloway et al. speculated that requiring students to adhere to the single loop entrance and exit rule of structured programming "puts an extra burden on memory and processing resources." [121, p. 854] The authors labeled the traditional structured loop solution as a PROCESS/READ, indicating that the current value would be processed then another value would be read before returning to the top of the loop. Notice that the value being processed is the value read in the *previous iteration* of the loop. The other option, leaving in the middle of the loop, was labeled READ/PROCESS. This meant that a value would be read and then processed if the value was not the sentinel. This format requires the ability to be able to exit from the middle of the loop (when the sentinel is found). Soloway et al. believed that because the READ and the PROCESS were in sync, meaning that the read and process of the value read occurred in the same iteration of the loop, it was easier cognitively for the students. Examples of both algorithms can be seen in Figure 32.

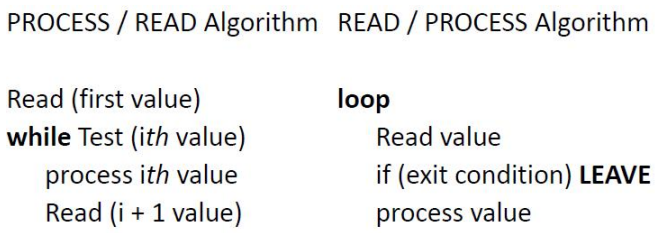

Figure 32: Traditional vs. Exit in the Middle Algorithm [121]

The authors conducted a study with novice, intermediate, and senior programmers. They asked the students to design a solution to the rainfall problem and then implement a solution. Half the students implemented the solution in traditional PASCAL while the other half implemented the solution in PASCAL L, a version of PASCAL with an additional command of LEAVE (similar to the break command in C).

All three populations had a strong preference for designing a solution that was the

READ/PROCESS (leave in the middle) where a strategy was discernible (80%). In the implementation phase, 73% used the READ/PROCESS (leave in the middle) strategy, regardless of whether they were in PASCAL or PASCAL L. More importantly was the program correctness. More people wrote a correct program using PASCAL L than did those using PASCAL, and the difference was statistically significant. In addition, all three levels of students benefited equally from PASCAL L.

Eric Roberts argued for limited use of the exit-in-the-middle [108]. He argued for the case in two limited conditions: the loop-and-a-half problem of sequential search and when reading input to be processed in a loop (i.e., the rainfall problem). This was at the time of moving away from PASCAL into C and made use of the break statement. However it should be noted that reviewers of his textbook complained against the practice of using a break statement within a loop, stating that it was "unacceptable" as it was similar to using the dreaded GOTO statement and violated the basic tenets of structured programming.

Barnes and Shinners-Kennedy set out to determine if student (and instructor) preference of using the exit in the middle had changed with the increase in object-oriented programming [15]. They gave a set of university academics and a group of instructors and post graduate students another loop-and-a-half problem originally put forth by Yuen [154]. They found that 50% of the first group and 60% of the second group produced solutions with break statements. Clearly the natural preference has not changed in 30 years.

This study was designed to test the hypotheses that using the exit in the middle strategy for learners would result in greater learning performance than using the traditional loop format. By changing the format of the loops that students use to learn will result in a lower cognitive load yielding more learning. Additionally, student performance on low cognitive load activities using the alternate loop format will also be explored. Students will be presented with two low cognitive load assessments (Parsons problems). One Parsons problem will contain a programming solution in the traditional loop format, the other will contain a programming solution with an exit in the middle format.

6.2 Study Method

This study is adapted from the subgoal labels study (Chapter 5). The original materials and process were used as the basis and adapted for the new intervention. This is a 2x2 study with the independent variables of worked examples and test question format. The dependent variable will be learning gains as measured through problem assessment tasks of generating code and Parsons problems.

6.2.1 Instructional Materials

The instructional materials used in this study were adapted from the subgoal labels study (Chapter 5). The training problem and each worked example were reworked to be in the form using an exit-in-the-middle approach, with appropriate subgoal labels. See Figure 33 for a portion of a worked example using the exit in the middle format.

Half of the participants saw this format of the worked examples while the other half of the participants saw worked examples with the traditional loop format. All of the worked examples contained subgoal labels. However the order of the subgoal labels differed between the exit in the middle group and the traditional loop group, as the solution process is slightly different. For this study, only the given subgoal label condition was used, meaning that all participants were given subgoal labels in all worked examples.

All of the worked example - practice pairs contained contextual transfer in this study. In the subgoal study (Chapter 5) the given-isomorphic group (those given subgoal labels who saw only isomorphic worked example - practice pairs) was the anomaly that performed poorly on the assessment tasks. In this study we provided all participants with contextual transfer between the worked example - practice pair. This decision was based on the data from the subgoal study, with the thought that the contextual differences may allow students to recognize the similarities between different contextual examples and possibly generalize across the examples. The alternative would have been to use all isomorphic problems between the worked examples and practice problems. I decided to use the contextual transfer

Worked Example #1 for Sub-Goal CS Study – Calculating Average

Your friend is working as a server in a restaurant. He has a collection of tips, but wants to know what his average tip is on a Friday night. You have volunteered to write a program for him to help him calculate his average tip.

Here are his tips for last Friday night. What is his average tip?
$15.00, $5.50, $6.75, $10.00, $12.00, $18.50, $11.75, $9.00

Solution

. . .

SUBGOAL: write loop

Step four: loop until you EXIT

```
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]
sum = 0
index = 0
WHILE  True DO

ENDWHILE
```

SUBGOAL: exit loop when condition met

Step five: if we are at the last tip, exit the loop

```
# If we are at the position equal to the length of the list, then we are past the end so leave the loop
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]
sum = 0
index = 0
WHILE  True DO
    if index == length (tips) THEN
        EXIT
    ENDIF

 ENDWHILE
```

SUBGOAL: update loop

Step six: move to the next tip

```
# We want to look at every element in the list so the update of index is by one
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]
sum = 0
index = 0
WHILE  True DO
    if index == length (tips) THEN
        EXIT
    ENDIF
    index = index + 1
ENDWHILE
```

Figure 33: Sample Problem using Exit in the Middle Format

condition because it performed the best in the subgoal study.

6.2.2 Assessment Materials

One consideration is that all of the current pre / post test materials used in the previous studies were developed using a traditional while loop format. To help determine if it is the format of the loop that makes the difference, equivalent test questions were developed using the exit in the middle format. Half of the pre / post assessment questions were altered to be in the exit in the middle format. See Table 17 for an example of one of the altered questions.

Table 17: Sample Pre Post Question

Traditional Question	**Exit in the Middle Question**

Consider the following code segment:
value = 15
WHILE value <28 DO
  PRINTLN value
  value = value + 1
ENDWHILE
 	
What are the first and last numbers output by the code segment?
first last
a. 15 27
b. 15 28
c. 16 27
d. 16 28
e. 16 29
	Consider the following code segment:
	value = 15
	WHILE True DO
	  IF value >= 28 THEN
	  EXIT
	  ENDIF
	  PRINTLN value
	  value = value + 1
	ENDWHILE
	
	What are the first and last numbers output by the code segment?
	first last
	a. 15 27
	b. 15 28
	c. 16 27
	d. 16 28
	e. 16 29

Rather than changing all of the questions, half of the questions were altered for half of the participants. Half of the subjects saw a pre \post test with only the traditional loop format and the other half saw mixed test questions.

Also, two different Parsons problems were used in this study. The first Parsons problem was the same as the one used in the original Subgoal Label study (in a traditional loop

format). The second Parsons Problem presented a solution using the exit in the middle solution. All participants saw both Parsons problems. For a complete picture of the 2x2, between subjects, factorial study design, see Table 18.

Table 18: Loop Strategy Study

		Worked Example Format	
		Traditional	**Exit in the Middle**
Pre / Post Test Content	**Traditional**	All traditional loops in pre/post test, worked examples (Replicates condition in subgoal label study)	All traditional loops in pre/post test; Exit in the middle format in worked examples.
	Exit in the Middle	Pre/post test will contain 1/2 of the questions with exit in the middle format. Worked examples will be traditional loop format.	Pre/post test will contain 1/2 of the questions with exit in the middle format. Exit in the middle format in worked examples

6.2.3 Protocol

The procedure for this study was similar to the subgoal label study (5). Specifically the steps were:

1. Consent
2. Demographics
3. Pre-test (in appropriate format)
4. Worked Example - Practice Problems (3 in appropriate format)
5. Cognitive Load measurement
6. Problem Solving Assessment - Parsons Problems (both formats presented)
7. Cognitive Load measurement
8. Problem Solving Assessment - writing code

9. Assessment - Post-test (in appropriate format)

The cognitive load measurement survey was repeated twice. The first time it measures the perceived cognitive load of the instruction. The second time it measures the perceived cognitive load of the Parsons problem assessments.

Throughout the study the time was recorded for each task. We collected process data and performance data on the practice problems to ensure that participants were actually completing the tasks.

6.2.4 Participants

Participants were recruited from introductory programming classes at Georgia Tech, KSU (formerly SPSU), and the University of Nebraska Omaha. Students were from a variety of introductory programming classes representing multiple majors and using different programming languages for instruction. See Table 19 for detailed information about the participants.

An effort was made to recruit students at the point in the semester where they had been exposed to loops but before they had been tested over loops. Since none of the instructors of the courses recruited from taught the exit in the middle format, it would have been new to the participants. However, many students reported previous experience with writing loops (GT 60%, KSU 70%, and UNO 90%). It is not known if this previous experience was within their current class or a previous programming course.

Table 19: Participant Demographics

School	**Georgia Tech**	**KSU/SPSU**	**UNOmaha**
Number	7	111	38
Incentive	None	Lab credit	$20
Age	M=18	M=22	M=21
Gender	28.5% male	77.5% male	81.6% male
HS GPA	M=4.02	M=3.42	M=3.60
College GPA	M=3.7	M=3.1	M=3.5
Years in College	M=1	M=2.3	M=2.1

Other demographic data collected included information about their prior programming experience in high school, reported comfort with solving programming problems, expected difficulty of the programming task and primary language. There were no statistical differences between the treatment groups for demographic data, which is expected because participants were randomly assigned to treatment groups. Participants took a multiple-choice pre-test to measure problem solving performance, and any participant scoring more than 50% correct were eliminated from analysis as the intervention was designed for novices. This eliminated 5 KSU students and 5 UNO students. Average scores on the pre-test (after elimination of those with a score of 4 or better) were low as can be seen in Table 20. Twenty-five percent of all the students earned 0 points on the pre-test.

Table 20: Pre-Test Scores

School	**Pre Test**
GT	M=1.71 (28.5%)
KSU/SPSU	M=1.17 (19.5%)
UNOmaha	M=2.03 (33.8%)

Compensation may have affected the participant motivation and response rates. Participants received compensation regardless of the amount of time or effort that they devoted to the experiment, which might have caused low motivation in some participants.

- Georgia Tech participants received no compensation and data collection was done completely over the internet with no direct contact with the researcher. Announcements were made in class by the instructor with directions to contact the researcher via email to participate. This may account for the extremely low response rate from Georgia Tech. It is interesting to note that more females than males volunteered to participate in the study.

- KSU participants were given lab credit for participation, regardless of how much of the study they completed. At KSU, the researcher was not present for the data collection in the labs, as was the case in the subgoal study. Instructors at KSU were

responsible for distributing the reference sheet and assigning participant numbers and determining when students were completed with the lab. This may account for the very low completion rate at KSU. Also, in the subgoal study, CS classes, Engineering classes, and Gaming classes were recruited and participated. In this study no CS classes agreed to participate in the study. It may be that the Engineering and Gaming students were less invested in either learning the material or improving computer science education.

- By far the best completion rate (almost 100%) occurred at UNO, likely due to the researcher's presence, the paid incentive ($20), and one instructor offered a small extra credit for participating. Interestingly, most students indicated they were more motivated by the extra credit rather than the monetary incentive.

Once participants who did not compete all tasks were removed, a total of 93 participants' data was analyzed.

6.3 Data Analysis

The participants' solutions for the assessment tasks, both open coding and the Parsons problems, were scored. Solutions were scored using the same method as in the original Subgoal Label study (Chapter 5). The participants earned one point for each correct line of code that they wrote. This scoring scheme allowed for more sensitivity than scoring solutions as wholly right or wrong. If participants wrote lines of code that were conceptually correct but contained typos or syntax errors (e.g., missing a parenthesis), they received points. We scored logic errors (having < rather than a <=) as incorrect. Conceptual and logical accuracy were considered more valuable than scoring for absolute syntactical accuracy as the participants were still early in the learning process. Participants could score a maximum score of 44.

6.4 Results

6.4.1 Problem Solving

The effect of the interventions on problem solving performance depended on the interaction of the worked example format and test format. There was no main effect of the worked example format found on problem solving, F(1,91) = 1.057, MSE = 262.58, p = .307. In addition, we found no main effect of the test question format on problem solving, F(1,91) = 0.003, MSE = 0.829, p = .954. There was, however, a statistically significant interaction between worked example format and test question format, F(1,89) = 4.75, MSE = 1144.986, p = 0.032, est. ω^2 = .0138, f = .226 (see Figure 34).

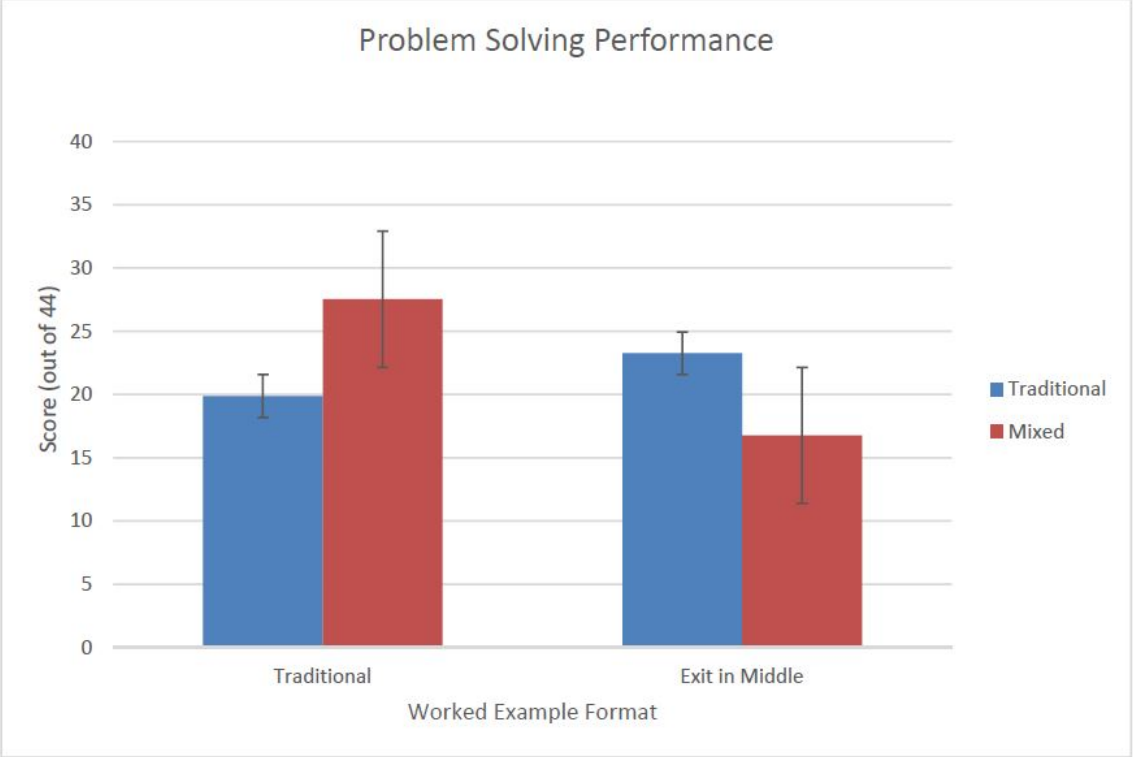

Figure 34: Problem Solving Performance based on Worked Example Format and Test Format

6.4.2 Parsons Problems

There was no statistically significant differences found on the Parsons problems solutions, either individually or combined (see Table 21).

Table 21: Parsons Problem Statistical Results

	Worked Example Format (WE)		**Test Format (T)**		**WE * T Interaction**	
Dependent Variable	**F test**	***p* value**	**F test**	***p* value**	**F test**	***p* value**
Traditional Parsons Problem	F(1, 107) = .622	.432	F(1, 107) = .009	.927	F(1, 105) = .967	.328
Exit in Middle Parsons Problem	F(1, 108) = .154	.696	F(1, 108) = 1.212	.273	F(1, 106) = 1.242	.268
Combined Parsons Problem Score	F(1, 107) = .135	.714	F(1,107) = .301	.584	F(1, 105) = 1.177	.280

6.4.3 Cognitive Load

There was no statistically significant differences found on the cognitive load measurements based on either the worked example format or the test question format. This was true for both the worked example cognitive load measure and the Parsons problem assessment measurement. See Table 22 for details.

Table 22: Cognitive Load Statistical Results

	Worked Example Format (WE)		**Test Format (T)**		**WE * T Interaction**	
Dependent Variable	**F test**	***p* value**	**F test**	***p* value**	**F test**	***p* value**
First Cognitive Load Measurement	F(1, 99) = .999	.320	F(1, 99) = .1.414	.237	F(1, 97) = .038	.846
Second Cognitive Load Measurement	F(1, 105) = .025	.875	F(1, 105) = .086	.770	F(1, 103) = .330	.268

6.4.4 Post Test

There was no statistically significant differences found on the performance on the post test based on worked example or test question format. See Table 23 for details.

Table 23: Post Test Statistical Results

	Worked Example Format		**Test Format**		**Worked Example * Test Format Interaction**	
Dependent Variable	**F test**	***p* value**	**F test**	***p* value**	**F test**	***p* value**
Post Test	F(1, 105) = .073	.788	F(1, 105) = .689	.409	F(1, 103) = .114	.736

6.4.5 Time

There was no statistically significant differences found on the time for completion in any section of the study based on worked example or test question format. See Table 24 for details.

Table 24: Time Statistics

	Worked Example Format (WE)		**Test Format (T)**		**WE * T Interaction**	
Dependent Variable	**F test**	***p* value**	**F test**	***p* value**	**F test**	***p* value**
Pre Test and Training Time	F(1, 123) = 1.094	.298	F(1, 121) = 1.107	.295	F(1, 119) = 1.089	.299
Worked Example and Practice Problem Time	F(1, 121) = .725	.396	F(1, 121) = .576	.449	F(1, 119) = .541	.464
Assessment and Post Test Time	F(1, 113) = .064	.800	F(1, 113) = .003	.960	F(1, 111) = 1.462	.229

6.5 Discussion

The initial hypotheses to be tested in this study were:

1. That students who learn using the exit in the middle format will do better on all of the assessment exercises (coding, Parsons problem, and post-test) than those who learn

the traditional loop.

2. All students will do better on the post test questions which are in the exit in the middle format.

3. Those who learn using the exit in the middle format will take less time in both the instruction and in the assessment tasks since it requires less cognitive processing.

4. Students who learn using the exit in the middle format will perceive it to have less of a cognitive load than the traditional format.

5. Those that receive the post-test questions using the exit in the middle format will perceive that to be a lower cognitive load than those who receive only the traditional questions.

The data does not support the first hypothesis. We have no evidence that students learning with the exit in the middle format perform better on any of the assessments. In fact, the only supporting data is that those who learned using the traditional loop format, but saw both loop formats on the pre-test performed statistically better than any other group. And those participants that learned with the exit in the middle format but saw the mixed question format for the pre test questions performed the worst on the open coding assessments.

There is no data to support hypotheses 2, 3, 4, or 5. No group did statistically better on the post test questions which were in the exit in the middle format. No group took statistically less or more time than any other group on any portion of the study. There was also no statistical difference in the cognitive load measurements based on the worked example format. Because there was no statistical difference in the cognitive load measurements on either the worked example format or the Parsons problems I have no reason to believe there would be a difference in the perception of the questions either.

6.6 Contributions

This study was designed to answer research question 3 of the dissertation:

RQ3 What is the effect on learning performance when teaching exiting loop in the middle versus traditional loops (single entry / exit)?

H3A: Exit in the middle loop will result in increased learning performance, on both low and high cognitive load assessments.

H3B: Assessment questions that use exit in the middle loop format will result in higher assessment scores regardless of instructional material format.

As we have no data supporting either hypotheses, the answer to the research question is that teaching loops using the exit in the middle strategy has no effect on learning performance. It does not appear that teaching students using the perceived more "natural" cognitive fit (loops that exit in the middle) improves learning performance.

CHAPTER VII

THE ROLE OF COGNITIVE LOAD IN LEARNING PROGRAMMING

This dissertation began with the following thesis statement:

> Learning how to program is an activity with a high cognitive load, which I theorize is due to its high intrinsic load. Learning programming and assessing the learned knowledge involves two separate tasks: the learning activity and the assessment activity, both of which typically involve high cognitive load programming tasks. Lowering the cognitive load in either the learning or assessment activity will result in more measurable learning gains.

To support the thesis statement, the dissertation sought to answer three research questions, which have been successfully addressed. Not all results agreed with the original hypotheses (see Table 25), but each does contribute to our knowledge of how cognitive load impacts learning programming.

7.1 What We've Learned

Given the five studies presented in this dissertation, this chapter is devoted to looking at the bigger picture and how each provides insights into the role of cognitive load in learning programming. The overarching goal behind this research was to gain more understanding and insight into when cognitive load might be interfering with learning programming. We know that many students find learning programming difficult and do not succeed in the current learning environments. Are there ways we can improve their chances of success in learning programming by using principles originally developed from within educational psychology and found to be successful in other disciplines?

Table 25: Research Questions and Results

Research Questions	**Hypotheses**	**Results**
RQ1: Does altering the modality (text, oral, both) of code explanations improve study learning as measured by retention and transfer questions?	**H1A**: Students receiving oral explanations will demonstrate better retention.	No supporting evidence found.
	H1B: Students receiving both oral and text explanations will demonstrate the worst retention.	No supporting evidence found.
RQ2: Does introducing subgoals (either given or learner generated) result in better learning performance?	**H2A**: Learning activities with subgoals result in better learning performance than those without subgoals.	Partially supported. Groups who learned with subgoal labels performed better on problem solving assessment, but no statistical difference on post-test.
	H2B: Students who generate subgoals exhibit better learning performance than those who are given subgoals.	Partially supported. Generate groups performed better on problem solving assessment, but not on the post-test or Parson problem assessment.
	H2C: A lower cognitive load assessment activity can provide evidence of learning that a high cognitive load assessment does not.	Supported.
RQ3: What is the effect on learning performance when teaching loops that exit in the middle versus traditional loops (single entry / single exit)?	**H3A**: Exit in the middle loops will result in increased learning performance on both low and high cognitive load assessments.	Not supported.
	H3B: Assessment questions that use exit in the middle loop format will result in higher assessment scores regardless of instructional material format.	Not supported.

This dissertation has explored several options for applying educational psychology principles about cognitive load and multiple modalities into the field of computer science, specifically introductory programming. And like much research, the findings are mixed. Some of our results showed that the principles from educational psychology held in computing. Other results were different than predicted by previous studies in other disciplines. First we examine each study in isolation, recapping the premise and expected results along

with its findings. We also explore what the study tells us about cognitive load theory and programming and how future studies in the area might provide new insights into understanding the unique challenges of learning programming. We then address how this dissertation's findings support other theories within computer science education research and we conclude with possibilities for future work.

7.2 Study Summaries

7.2.1 Cognitive Load Measurement

The first study within this dissertation adapted an existing survey tool that purports to measure the different components of cognitive load –the extraneous, intrinsic, and germane. The adaptation was successful in the measurement of perceived cognitive load for given lectures within an introductory programming class. However, several interpretation issues have arisen since the implementation of that study:

- Components of Cognitive Load - At the time of the study, most prominent researchers in the field, including the creators of the original survey, believed there to be three distinct components of cognitive load (extraneous, intrinsic, and germane). Since then, some researchers argue that there is no "germane" load specifically, only the existing resources within working memory that can be taxed from extraneous and / or intrinsic load [63, 68, 130]. Partially because there is no way to directly manipulate the germane load within an experiment, we can consider germane load to be part of intrinsic load. The load placed on working memory that is not directly related to the inter-connectivity of the elements is now considered germane learning.

  In fact, the original survey instrument and the adaptation presented within the dissertation found the weakest support for the germane load. Instead what can be argued is the direction of the loads, i.e., when extraneous load increased the germane load decreased.

- Components Not Additive - In the original Cognitive Load Theory, the three components (extraneous, intrinsic, and germane) were considered to be additive. Thus if the extraneous or intrinsic load was too high, there would be a shortage of available working memory resources for the germane load, or actual learning. Now that researchers consider germane to be part of the intrinsic load, the instructional materials that can be manipulated are those that cause too much extraneous or intrinsic load. There is a limit on working memory, and too much extraneous or intrinsic load can overload that resource causing learning to suffer. However, just because a specific instructional material is designed to reduce the extraneous load on a learner, this does not mean that the learning will automatically increase. There are many other factors that may also be associated with whether learning increases –student motivation, interest in the subject and self-regulation. What we do know is that certain instructional design can easily overload working memory thereby reducing the learning that can occur –even in a motivated and interested learner. While we cannot guarantee that well designed instructional material will always result in greater learning gains, we can design instructional material that does not overload working memory for the average student.

- No Baseline Reference - In all of the follow-on studies which used the cognitive load measurement tool, no statistical difference was found. This was true even though the intervention was specifically designed to reduce either the extraneous or intrinsic load of the material being taught. It may be that the intervention did not succeed in reducing the extraneous or intrinsic load. However, it may be that the survey instrument is not sensitive enough to measure the perceived differences with no baseline activity to compare against. The cognitive load measurement survey instrument that was adapted to introductory programming was validated using students' perception of class lectures. Students in a university setting attend lectures almost daily. Most have been taught in a lecture based format since entering the school system. Students

likely find it natural to perceive whether or not a lecture was easy or more difficult to understand –thus the survey instrument yielded statistically significant results.

However when a student is presented with new material to learn in a novel format, how can they ascertain whether their learning is difficult or easy –as compared to what? Especially since they have no tangible way of knowing exactly how much they have learned or if what they learned was accurate? The survey instrument was given immediately after the instructional period but before the testing period and no feedback was given to the participants to indicate if their practice problem submissions were correct or incorrect. A learner may indicate they thought the learning was easy, but their answers may indicate they have not learned the material correctly.

I would argue that when learning new material, especially in a novel format, that participants be given a base task to attempt first and then rate the intervention *with respect to* the base task. Then all cognitive load measurements would be relative to the baseline. In other words, learners can report if the intervention was perceived to be more or less difficult than the base task. In this case the magnitude of the change would also be relevant when comparing different interventions. Using a baseline activity for measuring cognitive load in a learning environment is a new idea, but using baselines to measure cognitive load is not new. It is a common occurrence when measuring cognitive load using physiological measurements [53, 51, 32].

Moreno argues that a major shortcoming in the cognitive load theory field is the lack of uniform operationalization and measurement of pivotal CLT constructs such as mental effort, difficulty, expertise, prior knowledge, and the different types of cognitive load. [89]. A valid, reliable means to measure cognitive load and the extraneous and intrinsic components are needed in order to reliably test worked example interventions.

7.2.2 Modality

In the modality study, we attempted to replicate work from other domains that found that multi-modal explanations (written and verbal) yielded better learning provided certain conditions were met (the material wasn't redundant, too long, etc.) However the results did not replicate for any of the three introductory programming examples. This may be because the study was flawed:

- I may have overestimated the amount of learning and memory capacity within the participants. All the participants recruited had no prior programming experience, they were complete novices. It was naive of me to think that by watching a single video that they would be able to learn somewhat complex programming skills.

- The participants may not have been particularly motivated to learn the material. All participation was completed over the internet. The only incentive were entries to a raffle for Amazon gift cards. There was no internal motivation to learn the material.

- It may be that learning programming is significantly different than other domains. Based on the follow-on studies, we now know that other educational psychology principles (i.e., subgoal labels) do not completely replicate in the programming domain. This may be another principle that does not replicate in the programming domain.

- Programming may be so highly intrinsic that even motivated students could not hold all the necessary information within working memory to accomplish learning. If programming is completely interconnected, then there is no reason to believe that just by watching a short video, with no active learning and feedback, that learners would be able to internalize the material.

This may explain why there was no effect even on the recall questions –direct recall from information in the video.

Multi-modal explanation may yet lead to better learning in introductory programming, but demonstrating that experimentally will take a more sophisticated experiment. I conclude that the examples chosen for the videos were too complicated for the learners at the time they participated in the study. Participants were recruited at the beginning of their introductory programming course, before they had been exposed to any of the concepts presented in the videos. Given knowledge about the difficulties students have even after being presented with the material in lecture, reading the textbook, and even perhaps programming assignments, why would a simple minutes-long video would lead to understanding? In CLT terms, the learners had not automated even the simplest of concepts such as assignment. Literally every term and concept was new knowledge being presented. There was no prior knowledge on which to construct meaningful schemas.

If I were to re-do the experiment, I would present exceptionally short, simplistic videos of the most elementary programming concepts (assignment, arithmetic operations, print statements, etc.) followed by recall and application questions. I would present very simple information that anyone, not just students in an introductory programming class, could learn. As students progress through the introductory programming class, short multi-modal videos could be developed for each programming concept –as an introduction, not as the final or only means of instruction. I believe this type of intervention would yield more promising results similar to those found in other disciplines.

7.2.3 Subgoal Labels

The subgoal label study within this dissertation succeeded in supporting the hypothesis that the participants did learn more when presented with instructional materials that either gave them or had them generate their own subgoal labels. Students who generated their own labels that were more abstract performed better than any other group [76]. However teaching students how to generate abstract subgoal labels and providing adequate feedback on generated subgoal labels may take more time than the instructor is willing to provide.

The Given label condition performed almost as well as the Generate label condition.

It is not necessarily surprising that the intervention did not result in "far" transfer. Indeed, as Moreno states, "It seems unlikely that far transfer effects can be found within the current research designs and paradigms, which are limited to brief, one-time interventions, and where students are presented with a minimum amount of isomorphic worked examples and only tested immediately after instruction." [89, p. 177]

Future research needs to explore the use of subgoal labels within introductory programming to beyond just learning indefinite while loops. Future research should include developing and testing the use of subgoal labels for a variety of programming constructs (assignment, selection, method writing, array usage, etc.). (See 7.3.2 for additional details.) By expanding the use of subgoal labels throughout the entire course we can begin to determine the overall impact of using subgoal labels beyond a single intervention.

It is also possible to explore other ways to use subgoal labels within the curriculum. The study presented within this dissertation used subgoal labels during the knowledge acquisition phase but did not present the subgoal labels at all during the assessment phase. What if we were to provide the subgoal labels during the open coding assessments or the Parsons problems? What effect would that have on student learning? My hypothesis is that by adding subgoal labels to the assessments, students would be able to demonstrate more learning.

There is also the question of the puzzling result found in the subgoal label study –why learners who were given subgoal labels with isomorphic worked example / practice problem pairs performed so poorly. This result was replicated in a follow-on study thus we believe this is an actual phenomenon that needs further investigation. Research should be done to determine the cause behind the phenomenon and how to mitigate the problem. I believe that the cause is the lack of different contextual examples for learners to be able to generalize from. To determine this, a study involving a think-aloud protocol could be

used to try and understand what learners are thinking during the worked example / practice problem pairs. By comparing the two given subgoal label conditions (isomorphic and contextual transfer) we may be able to uncover the reason behind the phenomenon. Until this research is completed, instructors should be careful to always create contextual transfer between the worked example and practice problem if subgoal labels are given to the learner.

7.2.3.1 Subgoal Labels vs. Plans

We know that worked examples and subgoal labels benefit the learning of novice programmers. But what happens as their knowledge and skill proceed towards the expertise level? At what point do the initial subgoal schemas become automated? Learning to program is accomplished by developing a series of abstractions. An absolute novice will need to parse an assignment statement by evaluating the expression on the right hand side of the assignment operator and determining its data type. They would then need to determine the data type of the variable on the right hand side of the assignment operator. Then they can process the actual assignment operation. Whereas a student who has automatized the assignment operation may look at the assignment statement and immediately understand its purpose. This is much like a beginning reader must sound out words until they are recognizable. The reader then proceeds to work on the meaning of sentences.

Is there a next level for subgoal labels as learners automate the basic programming construct schemas? Instead of breaking down writing a while loop into multiple subgoals, as was done in the subgoal label study, perhaps writing the while loop becomes a single subgoal in a larger set of subgoals towards the larger goal of solving a problem. This is what Soloway presented as plans [120]. Soloway defined as a plan as a set of abstractions, each of which captures the essential features of a class of problems, and corresponding solution programs.

A conscious decision was made when designing the subgoal labels for the study presented in this dissertation to have the subgoals support the overall goal of writing a loop. The subgoals are all related to the specific programming structure, not necessarily to solving a problem. At the early stages of computational literacy, writing the program is the problem, like writing a sentence is for the early writer. This is a departure from the subgoal labels presented in other disciplines, such as math. In math (or physics), solving a problem normally involves applying the same problem solution pattern (or subgoals) with a different set of numbers. Thus the subgoals remain the same and only the individual set of data values change. In other words, calculating the standard deviation of a set of numbers is an identical process regardless of the numbers involved.

When solving a programming problem, the programmer is *determining and writing the subgoals* for the problem solution. The subgoals *are* the problem solution so that the program can be run multiple times with different sets of data. In essence, when solving a programming problem, the learner is developing the subgoals of the problem solution and then translating them into code. The given subgoal labels for the study presented here are not the problem solution, they are the subgoals for translating the problem solution into a programming construct.

This inevitably raises the question of “What comes next?" After subgoal labels for programming constructs, is there a way to create subgoal labels for the problem solving process? In the second programming course, programming construct subgoal labels might incur an expertise reversal effect because the problem solving process is at the program level not the programming level. In other words, students are working to determine the actual problem solution, not translating a known problem solution into a programming language. What should the next subgoal labels look like for determining the problem solution? I hypothesize that these labels would be more general but look more similar to the subgoal labels that appear in other disciplines. The problem solving subgoal labels would be related to problem categories.

7.2.4 Parsons Problems as Assessment

As we reduce the cognitive load during learning by altering the instructional material we need to also think about reducing cognitive load during assessment. When a learner's cognitive load is exceeded during an assessment activity, they cannot possibly demonstrate their knowledge or skills to their maximum. In introductory computer programming, instructors often use multiple choice, short answer, and long answer questions, much like other disciplines. However the long answer questions that involve problem solving require students to begin with a "blank page" and engage in a means-ends analyses, searching their long term memory for the next needed step. This can easily result in cognitive overload.

This dissertation provides evidence that there are at least two possible ways to reduce this cognitive overload: using Parsons problems and subgoal labels. As demonstrated in Chapter 5, using a Parsons problem to assess student knowledge can yield a more sensitive measurement of student learning. Students who perhaps cannot complete a means-ends analysis for a problem (in fact, may not even be able to determine the first step, thus leaving the question blank), can, when given the solution steps to the problem, begin to put them in the correct order.

We also know that using subgoal labels helps to improve learning of while loops in introductory programming (5). In the subgoal label study, the coding assessments provided no hints or subgoal labels, students were required to begin with that "blank page". What learning gains would occur if the subgoal labels were repeated in the coding assessments? In other words, in the open coding assessments and Parsons problems, give the students the actual subgoal labels rather than requiring them to remember them. The solutions then have the outline (subgoal labels) and students only need to remember the actual code steps related to each subgoal label. My hypothesis is that learners could demonstrate more learning. Alternatively, what if statements of code were given; could students identify the appropriate subgoal step associated with those code statements? The idea of providing subgoal labels in the assessment problem is a form of fading scaffolding for learners; with

the thought that as the learners progress, those subgoal labels would be removed.

7.2.5 Loop Strategy

In the final study for this dissertation we attempted to reduce the cognitive load of learning loops by introducing a new strategy that previous research indicates was more natural or "intuitive" for students. The findings did not support this. It may be that the current study was flawed or that previous research was incorrect. It's not completely clear in the previous research at what point in their programming knowledge the participants were studied. Definitely in the [15] study the participants were graduate students and faculty, presumably in the expert stage, and yet they used the exit in the middle loop. It may be that as learners progress in their learning they find the exit in the middle loop easier to fall back on, especially in teaching examples (as opposed to production code). This is much like an expert writer jotting a letter or note to a friend in less stylistic terms than their published work.

We are not the only ones questioning the use of the exit in the middle loop strategy among novice programmers. Smith et al. [118], studied data within the Blackbox repository (a large-scale international repository for novice coding assignments) and found that non-standard control structures were used in 7% of the unique files analyzed. All 618,438 occurrences are not specifically exit in the middle loops, but this strategy was identified as a common pattern. In [122] the authors argue that using the exit in the middle strategy is common when doing non-input / output based programs (e.g., GUIs). Their argument is to go a step further and move onto plans and higher-order functions when teaching novices. The argument is supported with evidence in [112] which discusses research reporting on student performance on Soloway's Rainfall Problem. The only published report of "beating" the problem occurred with students using higher-order functions [43].

It is interesting to note that the group that performed the best on the learning assessments in the loop strategy study was the group with the traditional worked example format

but the pre and post questions were in the mixed format. This may be due to prior knowledge, assuming that the prior knowledge was based on the traditional format of the loop. But the performance difference may also be because the students actually learned from the pre-test. This is known as retrieval practice [117]. It has been argued that retrieval practice yields no benefit to the learner when learning complex material [50], yet Karpicke and Aue argue that it is possible [66]. Certainly additional research could be done to determine if retrieval practice applies to introductory programming.

7.2.6 Conclusion

All the studies in this dissertation, except the cognitive load measurement study, used worked examples. Using worked examples in novice problem-solving instruction is consistent with the well-known *ACT-R* framework [4], a four stage model of expertise. In this model, learners who are in the first stage of skill acquisition will attempt to solve problems by analogy. They use known examples of problems and try to relate those problems to the new problems to be solved. In the second stage, learners have developed schemas which guide them in future problem solving. At the third stage, provided the learner has performed sufficient practice, the schemas become proceduralized or automated, which leads to the fourth stage, expertise. In this fourth expertise stage, the automated schemas provide analogical reasoning on a large pool of examples. "Learning with worked examples is most important during the initial skill acquisition stages for well structured domains such as physics, programming, and mathematics." [89, p. 170]

However good worked examples do not guarantee learning will occur –they merely provide the best possible chance for learning to occur. Moreno argues that the cognitive load and learning effects resulting from different example designs and cognitive activities have been extensively researched; but what is clear is that the effects are highly dependent on students' characteristics [89]. She goes on to state that individual differences that are relevant to CLT should be taken into consideration when trying to derive theoretical

and practical implications for example-based learning. However, the learning benefits of example-based methods that prompt students to engage in essential cognitive processing are not only dependent on students' prior knowledge. Moreno proposes an alternative cognitive-affective theory of learning with media that includes self-regulation and motivation factors as learning mediators, CATLM [88]. More work is needed to adapt worked examples within introductory computing based on learner's prior knowledge and affective factors such as motivation and self-regulation.

7.3 Future Work

I plan to continue two lines of research in my future work: finding a more accurate and reliable measurement of cognitive load, especially as related to introductory programming, and expanding the use of subgoal labels in introductory programming.

7.3.1 More Accurate Measurement of Cognitive Load –Eye Tracking

Since the identification of CLT, researchers have searched for a means to measure cognitive load. To date, this has been accomplished through indirect, subjective, and direct measures (see 2. Indirect measures include learner performance indicators and error rates. Subjective measures consist of learners assessing the amount of mental effort or difficulty required during learning. Two basic means of measuring cognitive load through direct measures are using a dual task and physiological measurements. The advantage of direct measures of cognitive load is that it can provide an almost continuous measure of cognitive load during a task and is not prone to learner memory errors. Dual task measurements require learners to engage in an additional cognitive activity that is secondary to the primary task of learning, such as noticing a color change or auditory beep during learning. Physiological measures of cognitive load include heart rates, pupillary response, EEGs, and eye tracking. To date these types of measurements have been limited due to the expense and specialization of equipment. However, as with most hardware items, the prices, efficiency, and usability of the equipment now make it possible for smaller research labs to acquire the necessary

hardware and software to conduct this type of research.

As data processing has become more powerful, compact and affordable, eye tracking systems have followed suit. The result of these advancements has been the development of a diverse network of eye tracking applications that includes Human-Computer Interaction (HCI), neuroscience, psychology, education, medicine, defense, and many others. Researchers have known for decades that the behavior of the pupil reflects activity in the brain [60]. Early research suggested that simple changes in pupil size correlated with cognitive effort, but measurement of this association is difficult because of the influence of light on pupil size.

Researchers have now developed a technique for measuring cognitive workload based on changes in pupil diameter [78]. The Index of Cognitive Activity uses the signal processing techniques of wavelet analysis to detect small but reliable increases in pupil size while minimizing the impact of changes in light [79]. Consistently, difficult tasks and novices have higher workload than experts on the same task [80].

To more accurately determine a measurement based on eye tracking, the following research questions must be answered:

1. Does the Index of Cognitive Activity appropriately measure cognitive load for novice programmers?

2. Do current indirect, subjective, and physiological direct measures of cognitive load correlate for novice programmers?

3. Can the cognitive load of learning individual programming concepts be accurately measured? If so, can they be ranked by difficulty of learning?

In future work I intended to use the Index of Cognitive Activity and eye tracking to measure the cognitive load of learning to program. The first step will be to validate the Index of Cognitive Activity with simple programming tasks for both novices and experts. A group of participants, both novices and experts, will be recruited to solve simple programming

tasks involving mathematical calculations, selection statements, and loops. Participants will be asked to complete subjective measures (surveys) on the difficulty of solving each problem. Results of the indirect measure (performance on the task), subjective measure (self-assessed difficulty), and direct measures (eye gaze and pupil response) will be analyzed to determine if they agree and correlate with one another. Accuracy of each of the types of measures can also be assessed.

The next step will involve measuring the cognitive load of specific programming concepts. Cognitive Load Theory implies that once a student has learned a concept it is internalized, or automatized. Just as an experienced reader no longer looks at each individual letter of a word, experienced programmers can look at an assignment statement and understand it immediately without having to parse it out by each specific token. Because learning programming is cumulative, it builds on previous knowledge like mathematics, learners should comprehend and understand one concept before moving onto the next in order to prevent cognitive overload. The Index of Cognitive Activity and eye tracking will be used to determine if and when learners have "grasped" or internalized specific concepts. The longer a learner spends looking at and evaluating a statement indicates that they are still working on processing that statement and that it has not yet been internalized.

Novice programmers will be recruited to solve various programming tasks using specific programming concepts. Their eye gaze, pupil response, accuracy and time on task will be recorded, along with subjective measures. This will allow us to measure the cognitive load of learning each concept from all three perspectives in an effort to categorize the difficulty of learning each concept.

Completing this research will yield not only a calibrated, reliable means to measure cognitive load, it will also begin to measure the specific cognitive load associated with specific introductory programming tasks. While eye tracking alone will not measure the specific components of cognitive load, together with a base line activity and survey data, it may begin to shed light on how to measure the components separately.

7.3.2 Expand the Use of Subgoal Labels to the Entire Introductory Programming Course

The subgoal label study presented in this dissertation evaluated student learning of a single programming construct, indefinite while loops. There are numerous programming constructs in a typical introductory programming course taught with an imperative language. What learning gains could be realized if empirically tested worked examples with subgoal labels were available for all (or most) programming constructs within a course?

I intend to pursue a research line that involves designing and creating multiple worked examples with subgoal labels for a majority of programming constructs within an introductory programming course. I believe subgoal labels can be created for the assignment operator, selection statements, for (counting) loops, writing methods, writing classes, calling methods and parameter passing, and array processing. These worked examples could be empirically tested in classrooms using experiments similar to the ones in this dissertation. I would advocate to design simpler experiments that require less time on the part of the learner to improve completion and participation rates. Experiments could also occur throughout the term rather than as a single one-time intervention.

Additionally I want to create levels of scaffolding where the subgoal labels fade as the construct is automatized by the learner. In other words, students would begin with every assignment statement having subgoal labels. However, once the student automatizes the assignment statement and understands it as a single chunk, those subgoal labels will fade and the assignment operation will become part of a larger subgoal (such as initialization for a loop). Once subgoal labels for every programming construct are developed they can be embedded into multiple examples to demonstrate their generalization.

Another line of research is to use subgoal labels within assessments. Test questions can be developed where subgoal labels are given and students simply have to write the associated code with each subgoal. Or it can be reversed where the code is given and the student must identify the subgoal. Both of these approaches should involve a lower

cognitive load on the student allowing her to demonstrate more knowledge. To ensure that students are prepared for the next course, subgoal labels on assessments would need to fade as the course progresses. As we desire the learners to automatize the programming construct by internalizing the subgoal labels, this could be assessed by then removing the subgoal labels on assessment further into the teaching term. This is an alternate means to determine if the student has automatized the programming construct and no longer thinks in terms of individual subgoal labels.

Eventually subgoal labels could be incorporated into an Integrated Development Environment (IDE). The student could identify the programming construct to be used and the subgoal labels would appear within the source code as comments in the correct order. This would serve to remind the learner to write the associated code for each subgoal. Currently, subgoal labels within programming have only been tested using paper or text based (non-compiled) answers. How much learning would occur if the subgoals were presented *during* the coding process? How much learning time would be saved by prompting the novice programmer with all of the subgoals needed to correctly implement the programming structure? Eventually the subgoal labels would need to be customizable by the user; the user should be able to turn them on or off as desired, providing fading scaffolding for learning.

APPENDIX A

WORKED EXAMPLE

Subgoal Given group (B)

Your friend is working as a server in a restaurant. He has a collection of tips, but wants to know w
his average tip is on a Friday night. You have volunteered to write a program for him to help him
calculate his average tip.

Here are his tips for last Friday night. What is his average tip?
$15.00, $5.50, $6.75, $10.00, $12.00, $18.50, $11.75, $9.00

Solution

SUBGOAL: define and initialize variables

Step one: define and initialize variable to hold the collection of tips

```
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]        list containing all the tip value
```

Step two: define an initialize variable to hold the sum

```
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]        list containing all the tip value
sum = 0                                                  accumulator to hold sum of va
```

SUBGOAL: initialize the loop

Step three: initialize the loop to start at the beginning of the list of tips

```
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]
sum = 0
lcv = 0                                                  lcv is loop control variable
# We want to start by looking at the first value in the list which has an index of 0
```

SUBGOAL: determine loop condition

Sub-SUBGOAL: determine termination condition of loop

Step four: determine when we are done with the list of tips

```
# In this case we want to process every element in the list, so we will terminate wh
we reach the end of the list, or when lcv has the value of length(tips)
lcv >= length(tips)
```

Sub-SUBGOAL: invert the termination condition into a continuation condition

Step five: change the ending condition to be a continuing condition (continue looking at tips while…)

```
# if the termination is when lcv >= length(tips), then to reverse that we have:
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]
sum = 0
lcv = 0
WHILE  lcv < length(tips) DO

ENDWHILE
```

SUBGOAL: update loop

Step six: move to the next tip

```
# We want to look at every element in the list so the update of lcv is by one
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]
sum = 0
lcv = 0
WHILE  lcv < length(tips) DO

   lcv = lcv + 1
ENDWHILE
```

SUBGOAL: process body of loop (why did we write it?)

Step seven: add the current tip to the sum

```
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]
sum = 0
lcv = 0
WHILE  lcv < length(tips) DO
   sum = sum + tips[lcv]                    as lcv is incremented by one each
time through the loop, the next
   lcv = lcv + 1                            value in the list will be added to
sum
ENDWHILE
```

SUBGOAL: determine results

Step eight: calculate the average

```
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]
sum = 0
lcv = 0
WHILE  lcv < length(tips) DO
   sum = sum + tips[lcv]
   lcv = lcv + 1
ENDWHILE
average = sum / length(tips)                    the average is the sum of the elements
divided by the number of elements
```

Step nine: print results

```
tips = [15, 5.50, 6.75, 10, 12, 18.50, 11.75, 9]
sum = 0
lcv = 0
WHILE  lcv < length(tips) DO
   sum = sum + tips[lcv]
   lcv = lcv + 1
ENDWHILE
average = sum / length(tips)
PRINTLN average
```

REFERENCES

[1] Ahadi, A. and Lister, R., "Geek genes, prior knowledge, stumbling points and learning edge momentum: parts of the one elephant?," in *Proceedings of the ninth annual international ACM conference on International computing education research*, pp. 123–128, ACM, 2013. 4.1.1

[2] Anderson, J. R., "Acquisition of cognitive skill.," *Psychological review*, vol. 89, no. 4, p. 369, 1982. 2.1.1

[3] Anderson, J. R., *The architecture of cognition*. Psychology Press, 2013. 2.1.1

[4] Anderson, J. R., Fincham, J. M., and Douglass, S., "The role of examples and rules in the acquisition of a cognitive skill.," *Journal of experimental psychology: learning, memory, and cognition*, vol. 23, no. 4, p. 932, 1997. 7.2.6

[5] Antonenko, P. D. and Niederhauser, D. S., "The influence of leads on cognitive load and learning in a hypertext environment," *Computers in Human Behavior*, vol. 26, no. 2, pp. 140–150, 2010. 2.4.3, 2.4.4

[6] Atkinson, R. K., Derry, S. J., Renkl, A., and Wortham, D., "Learning from examples: Instructional principles from the worked examples research," *Review of educational research*, vol. 70, no. 2, pp. 181–214, 2000. 2.2.1, 2.5, 2.6, 5

[7] Atkinson, R. K., "Optimizing learning from examples using animated pedagogical agents.," *Journal of Educational Psychology*, vol. 94, no. 2, p. 416, 2002. 2.6

[8] Atkinson, R. K., Catrambone, R., and Merrill, M. M., "Aiding transfer in statistics: Examining the use of conceptually oriented equations and elaborations during subgoal learning.," *Journal of Educational Psychology*, vol. 95, no. 4, p. 762, 2003. 2.6

[9] Atkinson, R. K. and Derry, S. J., "Computer-based examples designed to encourage optimal example processing: A study examining the impact of sequentially presented, subgoal-oriented worked examples," in *Fourth International Conference of the Learning Sciences*, 2000. 2.6

[10] Ayres, P., "Using subjective measures to detect variations of intrinsic cognitive load within problems," *Learning and Instruction*, vol. 16, no. 5, pp. 389 – 400, 2006. 2.4.4

[11] Ayres, P. and Sweller, J., "Locus of difficulty in multistage mathematics problems.," *The American Journal of Psychology*, 1990. 2.4.1

[12] Ayres, P. L., "Systematic mathematical errors and cognitive load," *Contemporary Educational Psychology*, vol. 26, no. 2, pp. 227–248, 2001. 2.4.1

[13] Baddeley, A., *Working Memory*. Oxford University Press, 1986. 1, 2.2.3

[14] Baddeley, A., "Working memory," *Science*, vol. 255, no. 5044, pp. 556–559, 1992. 2.3

[15] Barnes, D. J. and Shinners-Kennedy, D., "A study of loop style and abstraction in pedagogic practice," in *Proceedings of the Thirteenth Australasian Computing Education Conference-Volume 114*, pp. 29–36, Australian Computer Society, Inc., 2011. 6, 6.1, 7.2.5

[16] Bennedsen, J. and Caspersen, M. E., "Failure rates in introductory programming," *ACM SIGCSE Bulletin*, vol. 39, no. 2, pp. 32–36, 2007. 1, 5

[17] Bjork, R. A., "Memory and metamemory considerations in the training of human beings.," in *Metacognition: Knowing about Knowing*, MIT Press, 1994. 5, 5.1.2

[18] Bransford, J., *How people learn: Brain, mind, experience, and school*. National Academies Press, 2000. 2.2, 2.5, 4

[19] Brown, Timothy, *Confirmatory Factor Analysis for Applied Research*. Guilford Press, 2006. 3.2, 3.3.1

[20] Brünken, R., Plass, J. L., and Leutner, D., "Direct measurement of cognitive load in multimedia learning," *Educational Psychologist*, vol. 38, no. 1, pp. 53–61, 2003. 2.4.3

[21] Cant, S. N., Jeffery, D. R., and Henderson-Sellers, B., "A conceptual model of cognitive complexity of elements of the programming process," *Information and Software Technology*, vol. 37, no. 7, pp. 351–362, 1995. 2.7.5, 4

[22] Caspersen, M. E. and Bennedsen, J., "Instructional design of a programming course: a learning theoretic approach," in *Proceedings of the third international workshop on Computing education research*, pp. 111–122, 2007. 2.7.2

[23] Catrambone, R., "Improving examples to improve transfer to novel problems," *Memory & Cognition*, vol. 22, no. 5, pp. 606–615, 1994. 2.6

[24] Catrambone, R., "Generalizing solution procedures learned from examples.," *Journal of Experimental Psychology: Learning, Memory, and Cognition; Journal of Experimental Psychology: Learning, Memory, and Cognition*, vol. 22, no. 4, p. 1020, 1996. 1.3.3, 2.6

[25] Catrambone, R., "The subgoal learning model: Creating better examples so that students can solve novel problems.," *Journal of Experimental Psychology: General*, vol. 127, no. 4, p. 355, 1998. 1.3.3, 2.6, 5, 5.4

[26] Catrambone, R., "Aiding subgoal learning: Effects on transfer.," *Journal of educational psychology*, vol. 87, no. 1, p. 5, 1995. 2.6

[27] Chandler, P. and Sweller, J., "Cognitive load theory and the format of instruction," *Cognition and instruction*, vol. 8, no. 4, pp. 293–332, 1991. 2.2.4, 2.4.1

[28] Chandler, P. and Sweller, J., "The split-attention effect as a factor in the design of instruction," *British Journal of Educational Psychology*, vol. 62, no. 2, pp. 233–246, 1992. 2.4.1

[29] Chandler, P. and Sweller, J., "Cognitive load while learning to use a computer program," *Applied cognitive psychology*, vol. 10, no. 2, pp. 151–170, 1996. 2.2.4, 2.4.3

[30] Chi, M. T., Bassok, M., Lewis, M. W., Reimann, P., and Glaser, R., "Self-explanations: How students study and use examples in learning to solve problems," *Cognitive science*, vol. 13, no. 2, pp. 145–182, 1989. 2.2.6, 2.5, 3.2, 5

[31] Cierniak, G., Scheiter, K., and Gerjets, P., "Explaining the split-attention effect: Is the reduction of extraneous cognitive load accompanied by an increase in germane cognitive load?," *Computers in Human Behavior*, vol. 25, no. 2, pp. 315–324, 2009. 2.4.4

[32] Cinaz, B., *Monitoring of cognitive load and cognitive performance using wearable sensing*. phdthesis, EidgenÃűssische Technische Hochschule ETH ZÃijrich, 2013. 7.2.1

[33] Clark, R. C. and Mayer, R. E., *E-learning and the science of instruction: Proven guidelines for consumers and designers of multimedia learning*. Pfeiffer, 2011. 3.3.3

[34] Clark, R. C., Nguyen, F., Sweller, J., and Baddeley, M., "Efficiency in learning: Evidence-based guidelines to manage cognitive load," *Performance Improvement*, vol. 45, no. 9, pp. 46–47, 2006. 2.2.6

[35] Cooper, G. and Sweller, J., "Effects of schema acquisition and rule automation on mathematical problem-solving transfer.," *Journal of educational psychology*, vol. 79, no. 4, p. 347, 1987. 2.5

[36] Cowan, N., "The magical mystery four how is working memory capacity limited, and why?," *Current directions in psychological science*, vol. 19, no. 1, pp. 51–57, 2010. 1, 4.4

[37] Cronbach, L. J., "How can instruction be adapted to individual differences," in *Learning and individual differences* (Gagne, R., ed.), pp. 23–39, Ohio, Merrill Books Columbus, 1967. 2.2.5

[38] Cronbach, L. J. and Snow, R. E., *Aptitudes and instructional methods: A handbook for research on interactions.* Irvington, 1977. 2.2.5

[39] DeLeeuw, K. E. and Mayer, R. E., "A comparison of three measures of cognitive load: Evidence for separable measures of intrinsic, extraneous, and germane load.," *J.\ of Educational Psychology*, vol. 100, no. 1, p. 223, 2008. 2.4.4

[40] Denny, P., Luxton-Reilly, A., and Simon, B., "Evaluating a new exam question: Parsons problems," in *Proceeding of the Fourth international Workshop on Computing Education Research*, pp. 113–124, ACM, 2008. 2.7.4

[41] Ehrlich, K. and Soloway, E., "An empirical investigation of the tacit plan knowledge in programming," in *Human factors in computer systems*, pp. 113–133, 1984. 2.1.2

[42] Eiriksdottir, E. and Catrambone, R., "Procedural instructions, principles, and examples how to structure instructions for procedural tasks to enhance performance, learning, and transfer," *Human Factors: The Journal of the Human Factors and Ergonomics Society*, vol. 53, no. 6, pp. 749–770, 2011. 2.5, 5, 5.1.2

[43] Fisler, K., "The recurring rainfall problem," in *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, (New York, NY, USA), pp. 35–42, ACM, 2014. 7.2.5

[44] Garner, S., "Reducing the cognitive load on novice programmers.," *ERIC*, 2002. 2.7.2

[45] Gerjets, P., Scheiter, K., and Catrambone, R., "Designing instructional examples to reduce intrinsic cognitive load: Molar versus modular presentation of solution procedures," *Instructional Science*, vol. 32, no. 1, pp. 33–58, 2004. 2.4.4

[46] Gerjets, P., Scheiter, K., and Catrambone, R., "Can learning from molar and modular worked examples be enhanced by providing instructional explanations and prompting self-explanations?," *Learning and Instruction*, vol. 16, no. 2, pp. 104–121, 2006. 2.4.4

[47] Ginns, P., "Meta-analysis of the modality effect," *Learning and Instruction*, vol. 15, no. 4, pp. 313–331, 2005. 2.2.3

[48] Ginns, P., "Integrating information: A meta-analysis of the spatial contiguity and temporal contiguity effects," *Learning and Instruction*, vol. 16, no. 6, pp. 511–525, 2006. 2.2.2

[49] Gobet, F. and Simon, H. A., "Expert chess memory: Revisiting the chunking hypothesis," *Memory*, vol. 6, no. 3, pp. 225–255, 1998. 4

[50] Gog, T. and Sweller, J., "Not new, but nearly forgotten: the testing effect decreases or even disappears as the complexity of learning materials increases," *Educational Psychology Review*, vol. 27, no. 2, pp. 247–264, 2015. 7.2.5

[51] Grassman, M., Vlemincx, E., von Leupoldt, A., Mittelstädt, J. M., and Van den Bergh, O., "Respiratory changes in response to cognitive load: A systematic review," *Neural Plasticity*, 2016. 7.2.1

[52] Gray, S., St Clair, C., James, R., and Mead, J., "Suggestions for graduated exposure to programming concepts using fading worked examples," in *Proceedings of the third international workshop on Computing education research*, pp. 99–110, ACM, 2007. 2.7.2

[53] Haapalainen, E., Kim, S., Forlizzi, J. F., and Dey, A. K., "Psycho-physiological measures for assessing cognitive load," in *Proceedings of the 12th ACM international conference on Ubiquitous computing*, pp. 301–310, ACM, 2010. 7.2.1

[54] Hansen, M. E., Lumsdaine, A., and Goldstone, R. L., "An experiment on the cognitive complexity of code," in *Proceedings of the Thirty-Fifth Annual Conference of the Cognitive Science Society*, 2013. 4

[55] Hansen, M. E., Lumsdaine, A., and Goldstone, R. L., "Cognitive architectures: A way forward for the psychology of programming," in *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, pp. 27–38, ACM, 2012. 2.7.5, 2.7.5.1

[56] Hart, S. G. and Staveland, L. E., "Development of NASA-TLX (task load index): Results of empirical and theoretical research," *Advances in psychology*, vol. 52, pp. 139–183, 1988. 2.4.4

[57] Hu, Li-tze and Bentler, Peter M., "Cutoff criteria for fit indexes in covariance structure analysis: Conventional criteria versus new alternatives," *Structural Equation Modeling: A Multidisciplinary Journal*, vol. 6, no. 1, pp. 1–55, 1999. 3.3.1

[58] Jackson, Dennis, L., Gillaspy, J. Arthur, and Purc-Stephenson, Rebecca, "Reporting practices in confirmatory factor analysis: An overview and some recommendations," *Psychological Methods*, vol. 14, no. 1, pp. 6–23, 2009. 3.2

[59] Johnson, W. L. and Soloway, E., "PROUST: Knowledge-based program understanding," *Software Engineering, IEEE Transactions on*, no. 3, pp. 267–275, 1985. 5.1.5

[60] Kahneman, D. and Beatty, J., "Pupil diameter and load on memory," *Science*, vol. 154, no. 3756, pp. 1583–1585, 1966. 7.3.1

[61] Kalyuga, S., "When using sound with a text or picture is not beneficial for learning," *Australasian Journal of Educational Technology*, vol. 16, no. 2, 2000. 4.4

[62] Kalyuga, S., "Expertise reversal effect and its implications for learner-tailored instruction," *Educational Psychology Review*, vol. 19, no. 4, pp. 509–539, 2007. 2.2.5, 5.7.0.4

[63] Kalyuga, S., "Cognitive load theory: How many types of load does it really need?," *Educational Psychology Review*, vol. 23, no. 1, pp. 1–19, 2011. 2.2, 2.2.8, 7.2.1

[64] Kalyuga, S., "Instructional benefits of spoken words: A review of cognitive load factors," *Educational Research Review*, vol. 7, no. 2, pp. 145–159, 2012. 2.2.3, 4.4

[65] Kalyuga, S., Chandler, P., and Sweller, J., "Managing split-attention and redundancy in multimedia instruction," *Applied cognitive psychology*, vol. 13, no. 4, pp. 351–371, 1999. 2.2.4, 4

[66] Karpicke, J. D. and Aue, W. R., "The testing effect is alive and well with complex materials," *Educational Psychology Review*, vol. 27, no. 2, pp. 317–326, 2015. 7.2.5

[67] Kirschner, F., Kester, L., and Corbalan, G., "Cognitive load theory and multimedia learning, task characteristics, and learning engagement: The current state of the art," in *Third International Cognitive Load Theory Conference*, 2010. 1

[68] Kuldas, S., Hashim, S., Ismail, H. N., and Bakar, Z. A., "Reviewing the role of cognitive load, expertise level, motivation, and unconscious processing in working memory performance," *International Journal of Educational Psychology*, vol. 4, no. 2, pp. 142–169, 2015. 2.2, 7.2.1

[69] Leahy, W. and Sweller, J., "Cognitive load theory, modality of presentation and the transient information effect," *Applied Cognitive Psychology*, vol. 25, no. 6, pp. 943–951, 2011. 4.4

[70] Lee, M. J., Ko, A. J., and Kwan, I., "In-game assessments increase novice programmers' engagement and level completion speed," in *Proceedings of the ninth annual international ACM conference on International computing education research*, pp. 153–160, ACM Press, 2013. 5.1.1, 6

[71] Leppink, J., Paas, F., Van der Vleuten, C. P., Van Gog, T., and Van Merriënboer, J. J., "Development of an instrument for measuring different types of cognitive load," *Behavior research methods*, vol. 45, no. 4, pp. 1058–1072, 2013. 1.3.1, 2.2, 2.4.4, 3, 3.1, 3.2, 3.3.1, 3.3.2, 5.1.2

[72] Leppink, J., Paas, F., van Gog, T., van der Vleuten, C. P., and van Merriënboer, J. J., "Effects of pairs of problems and examples on task performance and different types of cognitive load," *Learning and Instruction*, vol. 30, pp. 32–42, 2014. 2.4.4, 3, 3.1, 3.4, 5

[73] Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., and others, "A multi-national study of reading and tracing skills in novice programmers," in *ACM SIGCSE Bulletin*, vol. 36, pp. 119–150, ACM, 2004. 5.7.0.5

[74] Margulieux, L. E. and Catrambone, R., "Improving problem solving performance in computer-based learning environments through subgoal labels," in *Proceedings of the first ACM conference on Learning@ scale conference*, pp. 149–150, ACM, 2014. 2.6

[75] Margulieux, L. E., Guzdial, M., and Catrambone, R., "Subgoal-labeled instructional material improves performance and transfer in learning to develop mobile applications," in *Proceedings of the ninth annual international conference on International computing education research*, pp. 71–78, ACM, 2012. 2.7.3, 5.4

[76] Margulieux, L. E., Morrison, B. B., Catrambone, R., and Guzdial, M., "Training Learners to Self -Explain: Designing Instructions and Examples to Improve Problem Solving," in *Transforming Learning, Empowering Learners: The International Conference of the Learning Sciences (ICLS) 2016*, vol. 1, (Singapore), pp. 98–105, June 2016. 7.2.3

[77] Margulieux, L. E., *Subgoal Labeled Instructional Text and Worked Examples in STEM Education*. phdthesis, Georgia Institute of Technology, 2014. 2.6, 5

[78] Marshall, S. P., "Method and apparatus for eye tracking and monitoring pupil dilation to evaluate cognitive activity," 07 2000. US 6,090,051. 7.3.1

[79] Marshall, S. P., "The index of cognitive activity: Measuring cognitive workload," in *Human factors and power plants, 2002. proceedings of the 2002 IEEE 7th conference on*, pp. 7–5, IEEE, 2002. 7.3.1

[80] Marshall, S. P., "Measuring Cognitive Workload in Simulation Environments," *8th Annual STISIM Users Group Meeting*, 2010. 7.3.1

[81] Mason, R. and Cooper, G., "Why the bottom 10% just can't do it: mental effort measures and implication for introductory programming courses," in *Proceedings of the Fourteenth Australasian Computing Education Conference-Volume 123*, pp. 187–196, 2012. 2.7.1

[82] Mason, R., Cooper, G., and de Raadt, M., "Trends in introductory programming courses in australian universities: languages, environments and pedagogy," in *Proceedings of the Fourteenth Australasian Computing Education Conference-Volume 123*, pp. 33–42, Australian Computer Society, Inc., 2012. 2.7.1

[83] Mayer, R. E., "Multimedia learning," *Psychology of Learning and Motivation*, vol. 41, pp. 85–139, 2002. 4, 4

[84] Mayer, R. E., *Multi-Media Learning*. Cambridge Univ Press, 2nd ed., 2009. 2.2.3, 4

[85] McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., Laxer, C., Thomas, L., Utting, I., and Wilusz, T., "A multi-national, multi-institutional study of assessment of programming skills of first-year CS students," in *Working group reports from ITiCSE on Innovation and technology in computer science education*, (Canterbury, UK), pp. 125–180, ACM, 2001. 5.7, 5.7.0.5

[86] Miller, G. A., "The magical number seven, plus or minus two: some limits on our capacity for processing information.," *Psychological review*, vol. 63, no. 2, p. 81, 1956. 1, 2.3

[87] Moreno, R., "Decreasing cognitive load for novice students: Effects of explanatory versus corrective feedback in discovery-based multimedia," *Instructional science*, vol. 32, no. 1, pp. 99–113, 2004. 2.4.2

[88] Moreno, R., "Instructional technology: Promise and pitfalls," *Technology-based education: Bringing researchers and practitioners together*, pp. 1–19, 2005. 7.2.6

[89] Moreno, R., "When worked examples don't work: Is cognitive load theory at an Impasse?," *Learning and Instruction*, vol. 16, no. 2, pp. 170–181, 2006. 7.2.1, 7.2.3, 7.2.6

[90] Moreno, R. and Mayer, R., "Interactive multimodal learning environments," *Educational Psychology Review*, vol. 19, no. 3, pp. 309–326, 2007. 2.2

[91] Morrison, B. B., Dorn, B., and Guzdial, M., "Measuring cognitive load in introductory CS: adaptation of an instrument," in *Proceedings of the tenth annual conference on International computing education research*, pp. 131–138, ACM, 2014. 5.1.5, 5.4

[92] Mousavi, S. Y., Low, R., and Sweller, J., "Reducing cognitive load by mixing auditory and visual presentation modes.," *Journal of educational psychology*, vol. 87, no. 2, p. 319, 1995. 4

[93] Nunnally, J. C., *Psychometric theory*. McGraw-Hill, 2nd ed., 1978. 3.3.1

[94] Paas, F., Tuovinen, J. E., Tabbers, H., and Van Gerven, P. W., "Cognitive load measurement as a means to advance cognitive load theory," *Educational psychologist*, vol. 38, no. 1, pp. 63–71, 2003. 2.4.2

[95] Paas, F. G., "Training strategies for attaining transfer of problem-solving skill in statistics: A cognitive-load approach.," *J.\ of educational psychology*, vol. 84, no. 4, p. 429, 1992. 2.4.2, 2.4.4, 2.5

[96] Paas, F. G. and Van Merriënboer, J. J., "The efficiency of instructional conditions: An approach to combine mental effort and performance measures," *Human Factors: The Journal of the Human Factors and Ergonomics Society*, vol. 35, no. 4, pp. 737–743, 1993. 2.4.2

[97] Paas, F. G. and Van Merriënboer, J. J., "Variability of worked examples and transfer of geometrical problem-solving skills: A cognitive-load approach.," *Journal of educational psychology*, vol. 86, no. 1, p. 122, 1994. 2.4.2, 2.5

[98] Paas, F. G., Van Merriënboer, J. J., and Adam, J. J., "Measurement of cognitive load in instructional research," *Perceptual and motor skills*, vol. 79, no. 1, pp. 419–430, 1994. 2.4.2, 2.4.3, 4

[99] Palmiter, S. and Elkerton, J., "Animated demonstrations for learning procedural computer-based tasks," *Human-Computer Interaction*, vol. 8, no. 3, pp. 193–216, 1993. 5, 5.1.2

[100] Pane, J. F., Myers, B. A., and others, "Studying the language and structure in non-programmers' solutions to programming problems," *International Journal of Human-Computer Studies*, vol. 54, no. 2, pp. 237–264, 2001. 6.1

[101] Parsons, D. and Haden, P., "Parson's programming puzzles: A fun and effective learning tool for first programming courses," in *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52*, ACE '06, pp. 157–163, Australian Computer Society, Inc., 2006. 2.7.4

[102] Peterson, L. and Peterson, M. J., "Short-term retention of individual verbal items.," *Journal of experimental psychology*, vol. 58, no. 3, p. 193, 1959. 1, 2.2.8

[103] Pirolli, P. and Recker, M., "Learning strategies and transfer in the domain of programming," *Cognition and instruction*, vol. 12, no. 3, pp. 235–275, 1994. 2.7

[104] Plass, J. L., Moreno, R., and Brünken, R., *Cognitive load theory*. Cambridge University Press, 2010. 2.2, 2.3

[105] Recker, M. M. and Pirolli, P., "Modeling individual differences in students' learning strategies," *The Journal of the Learning Sciences*, vol. 4, no. 1, pp. 1–38, 1995. 2.7

[106] Renkl, A. and Atkinson, R. K., "Structuring the transition from example study to problem solving in cognitive skill acquisition: A cognitive load perspective," *Educational psychologist*, vol. 38, no. 1, pp. 15–22, 2003. 5

[107] Rist, R. S., "Schema creation in programming," *Cognitive Science*, vol. 13, no. 3, pp. 389–414, 1989. 2.1.2

[108] Roberts, E. S., "Loop exits and structured programming: reopening the debate," in *ACM SIGCSE Bulletin*, vol. 27, pp. 268–272, ACM, 1995. 6.1

[109] Salomon, G., "Television is" easy" and print is" tough": The differential investment of mental effort in learning as a function of perceptions and attributions.," *J.\ of educational psychology*, vol. 76, no. 4, p. 647, 1984. 2.4.4

[110] Schnotz, W., "An integrated model of text and picture comprehension," *The Cambridge handbook of multimedia learning*, pp. 49–69, 2005. 2.2.3

[111] Schnotz, W. and Kürschner, C., "A reconsideration of cognitive load theory," *Educational Psychology Review*, vol. 19, no. 4, pp. 469–508, 2007. 2.2

[112] Seppälä, O., Ihantola, P., Isohanni, E., Sorva, J., and Vihavainen, A., "Do we know how difficult the rainfall problem is?," in *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, pp. 87–96, ACM, 2015. 7.2.5

[113] Shneiderman, B. and Mayer, R., "Syntactic/semantic interactions in programmer behavior: A model and experimental results," *International Journal of Computer & Information Sciences*, vol. 8, no. 3, pp. 219–238, 1979. 2.7.5

[114] Simon, H. and Chase, W., "Skill in chess," in *Computer chess compendium*, pp. 175–188, Springer, 1988. 1

[115] Simon, H. A., "How big is a chunk," *Science*, vol. 183, no. 4124, pp. 482–488, 1974. 1

[116] Skudder, B. and Luxton-Reilly, A., "Worked examples in computer science," in *Proceedings of the Sixteenth Australasian Computing Education Conference-Volume 148*, pp. 59–64, Australian Computer Society, Inc., 2014. 2.7.2

[117] Smith, M. A. and Karpicke, J. D., "Retrieval practice with short-answer, multiple-choice, and hybrid tests," *Memory*, vol. 22, no. 7, pp. 784–802, 2014. 7.2.5

[118] Smith, S. D., Zemljic, N., and Petersen, A., "Modern goto: novice programmer usage of non-standard control flow," in *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, pp. 171–172, ACM, 2015. 7.2.5

[119] Soloway, E. and Ehrlich, K., "Empirical studies of programming knowledge," *Software Engineering, IEEE Transactions on*, no. 5, pp. 595–609, 1984. 5.7.0.5

[120] Soloway, E., "From problems to programs via plans: The content and structure of knowledge for introductory LISP programming," *Journal of Educational Computing Research*, vol. 1, no. 2, pp. 157–172, 1985. 7.2.3.1

[121] Soloway, E., Bonar, J., and Ehrlich, K., "Cognitive strategies and looping constructs: An empirical study," *Communications of the ACM*, vol. 26, no. 11, pp. 853–860, 1983. (document), 6, 6.1, 32

[122] Sorva, J. and Vihavainen, A., "Break statement considered," *ACM Inroads*, vol. 7, no. 3, pp. 36–41, 2016. 7.2.5

[123] Spanjers, I. A., Van Gog, T., and Merriënboer, J. J., "Segmentation of worked examples: Effects on cognitive load and learning," *Applied Cognitive Psychology*, vol. 26, no. 3, pp. 352–358, 2012. 5

[124] Steele, C. M. and Aronson, J., "Stereotype threat and the intellectual test performance of african americans.," *Journal of personality and social psychology*, vol. 69, no. 5, p. 797, 1995. 4.1.2

[125] Steele, C. M., Spencer, S. J., and Aronson, J., "Contending with group image: The psychology of stereotype and social identity threat," *Advances in experimental social psychology*, vol. 34, pp. 379–440, 2002. 4.1.2

[126] Sweller, J. and Cooper, G. A., "The use of worked examples as a substitute for problem solving in learning algebra," *Cognition and Instruction*, vol. 2, no. 1, pp. 59–89, 1985. 2.5

[127] Sweller, J., "Cognitive load during problem solving: Effects on learning," *Cognitive science*, vol. 12, no. 2, pp. 257–285, 1988. 1.1, 2.2, 2.4.1, 2.5

[128] Sweller, J., "Cognitive load theory, learning difficulty, and instructional design," *Learning and instruction*, vol. 4, no. 4, pp. 295–312, 1994. 2.2

[129] Sweller, J., *Instructional Design in Technical Areas. Australian Education Review, No. 43.* ERIC, 1999. 2.2.2

[130] Sweller, J., "Element interactivity and intrinsic, extraneous, and germane cognitive load," *Educational psychology review*, vol. 22, no. 2, pp. 123–138, 2010. 2.2, 2.2.7, 2.6, 5.1.2, 7.2.1

[131] Sweller, J., Ayres, P., and Kalyuga, S., *Cognitive load theory*, vol. 1. Springer, 2011. 2.2, 2.2.1, 2.2.3, 2.2.4, 2.2.5, 2.2.6, 2.2.7, 2.2.8, 2.4.1, 2.4.2, 2.5, 4, 5

[132] Sweller, J. and Chandler, P., "Why some material is difficult to learn," *Cognition and instruction*, vol. 12, no. 3, pp. 185–233, 1994. 2.2, 2.2.4, 2.2.7

[133] Sweller, J., Van Merriënboer, J. J., and Paas, F. G., "Cognitive architecture and instructional design," *Educational psychology review*, vol. 10, no. 3, pp. 251–296, 1998. 2.2

[134] Tew, A. E. and Guzdial, M., "The FCS1: a language independent assessment of CS1 knowledge," in *Proceedings of the 42nd ACM technical symposium on Computer science education*, pp. 111–116, ACM, 2011. 5.1.1

[135] the Brain Unscrambles Jumbled Letters", H., "Why people can raed this, 4nd 7h15 | why people can read jumbled words and numbers in place of letters," tech. rep., livescience, 2015. 2.1.1

[136] Tindall-Ford, S., Chandler, P., and Sweller, J., "When two sensory modes are better than one.," *Journal of experimental psychology: Applied*, vol. 3, no. 4, p. 257, 1997. 4

[137] Trafton, J. G. and Reiser, B. J., "Studying examples and solving problems: Contributions to skill acquisition," tech. rep., Citeseer, 1993. 2.5

[138] Tuovinen, J. E. and Sweller, J., "A comparison of cognitive load associated with discovery learning and worked examples.," *Journal of educational psychology*, vol. 91, no. 2, p. 334, 1999. 2.2.5

[139] Underwood, G., Jebbett, L., and Roberts, K., "Inspecting pictures for information to verify a sentence: Eye movements in general encoding and in focused search," *Quarterly Journal of Experimental Psychology Section A*, vol. 57, no. 1, pp. 165–182, 2004. 2.4.3

[140] Van Gerven, P. W., Paas, F., Van Merriënboer, J. J., and Schmidt, H. G., "Memory load and the cognitive pupillary response in aging," *Psychophysiology*, vol. 41, no. 2, pp. 167–174, 2004. 2.4.3

[141] van Gerven, P. W., Paas, F., van Merriënboer, J. J., and Schmidt, H. G., "Modality and variability as factors in training the elderly," *Applied cognitive psychology*, vol. 20, no. 3, pp. 311–320, 2006. 2.4.3

[142] Van Gog, T. and Paas, F., "Instructional efficiency: Revisiting the original construct in educational research," *Educational Psychologist*, vol. 43, no. 1, pp. 16–26, 2008. 2.4.2

[143] van Gog, T. and Paas, F., "Cognitive load measurement," in *Encyclopedia of the Sciences of Learning*, pp. 599–601, Springer, 2012. 2.2

[144] van Gog, T. and Scheiter, K., "Eye tracking as a tool to study and enhance multimedia learning," *Learning and Instruction*, vol. 20, no. 2, pp. 95–99, 2010. 2.4.3

[145] Van Merriënboer, J. J. and De Croock, M. B., "Strategies for computer-based programming instruction: Program completion vs. program generation," *Journal of Educational Computing Research*, vol. 8, no. 3, pp. 365–394, 1992. 2.7.2

[146] Van Merriënboer, J. J. and Kirschner, P. A., *Ten steps to complex learning: A systematic approach to four-component instructional design*. Routledge, 2012. 1

[147] Van Merriënboer, J. J., Kirschner, P. A., and Kester, L., "Taking the load off a learner's mind: Instructional design for complex learning," *Educational psychologist*, vol. 38, no. 1, pp. 5–13, 2003. 1.1, 2.1.2

[148] Van Merriënboer, J. J. and Krammer, H. P., "The "completion strategy" in programming instruction: Theoretical and empirical support," *Research on instruction*, pp. 45–61, 1990. 2.7.2

[149] Van Merriënboer, J. J. and Paas, F. G., "Automation and schema acquisition in learning elementary computer programming: Implications for the design of practice," *Computers in Human Behavior*, vol. 6, no. 3, pp. 273–289, 1990. 1.1, 2.1.1, 2.1.2, 2.1.3, 2.5

[150] van Merriënboer, J. J. and Sweller, J., "Cognitive load theory and complex learning: Recent developments and future directions," *Educational psychology review*, vol. 17, no. 2, pp. 147–177, 2005. 2.2

[151] Vygotsky, L., *Mind in society*. Harvard University Press, 1978. 2.2

[152] Whelan, R. R., "Neuroimaging of cognitive load in instructional multimedia," *Educational Research Review*, vol. 2, no. 1, pp. 1–12, 2007. 2.4.3

[153] Wong, A., Leahy, W., Marcus, N., and Sweller, J., "Cognitive load theory, the transient information effect and e-learning," *Learning and Instruction*, vol. 22, no. 6, pp. 449–457, 2012. 4.4

[154] Yuen, C. K., "Programming the premature loop exit: From functional to navigational," *SIGPLAN Not.*, vol. 29, no. 3, pp. 23–27, 1994-03. 6.1